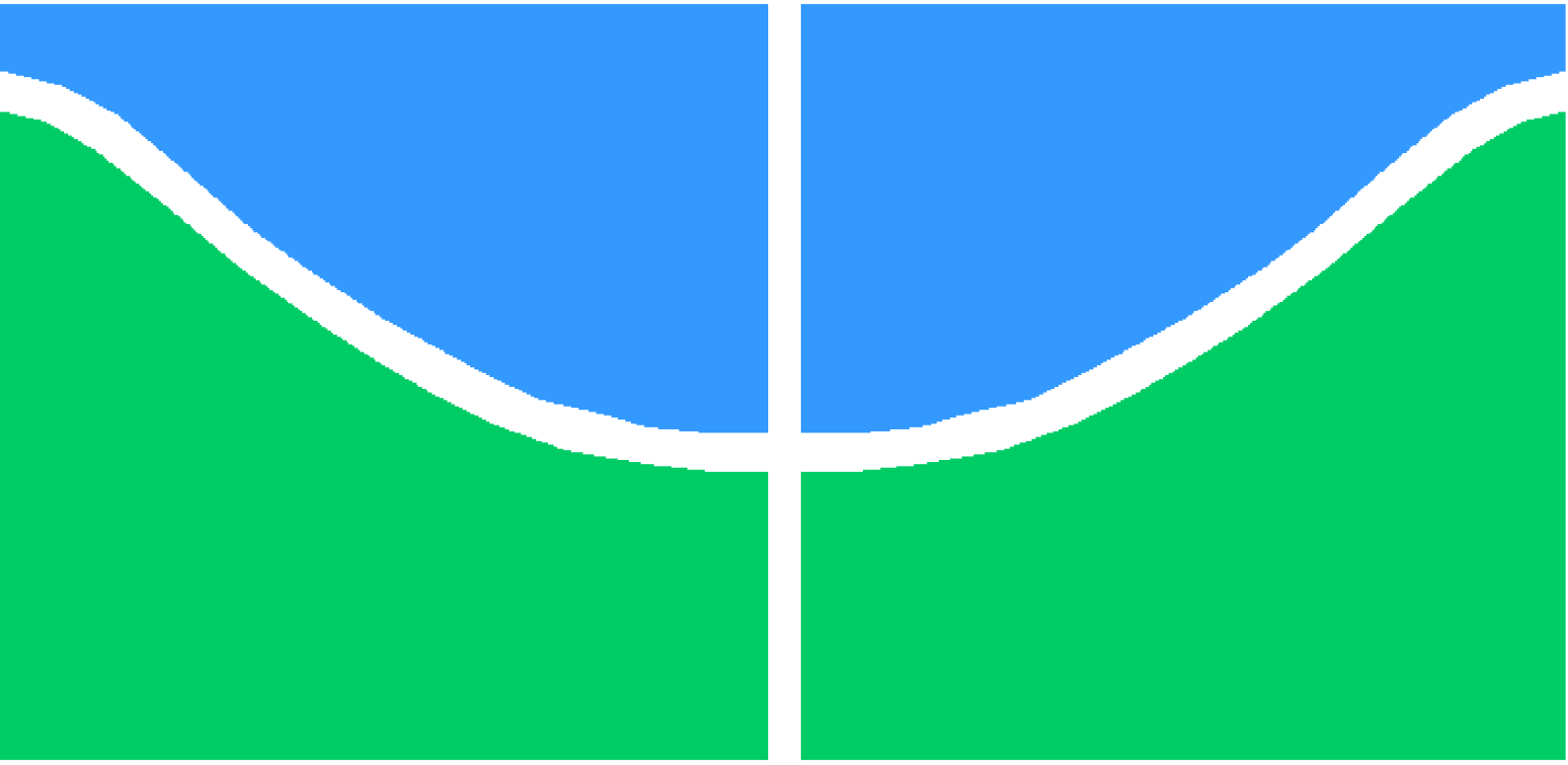

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia Eletrônica

Verificação Funcional de Modelos Transacionais de Processadores

Autor: Jair Dias de Oliveira Junior
Orientador: Prof. Dr. Gilmar Silva Beserra

Brasília, DF
2014

Jair Dias de Oliveira Junior

Verificação Funcional de Modelos Transacionais de Processadores

Trabalho de Conclusão de Curso submetido ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Gilmar Silva Beserra

Coorientador: Prof. Me. Tiago Trindade da Silva

Brasília, DF

2014

Jair Dias de Oliveira Junior
Verificação Funcional de Modelos Transacionais de Processadores/ Jair Dias de Oliveira Junior. – Brasília, DF, 2014-
136 p. : il. ; 29,7 cm.

Orientador: Prof. Dr. Gilmar Silva Beserra

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. Verificação Funcional. 2. Processador TLM-2.0. I. Prof. Dr. Gilmar Silva Beserra. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Verificação Funcional de Modelos Transacionais de Processadores

CDU 02:141:005.6

Jair Dias de Oliveira Junior

Verificação Funcional de Modelos Transacionais de Processadores

Trabalho de Conclusão de Curso submetido ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, DF, 24 de junho de 2014:

Prof. Dr. Gilmar Silva Beserra
Orientador

Prof. Me. Tiago Trindade da Silva
Convidado 1

Prof. Me. José Edil Guimarães de Medeiros
Convidado 2

Brasília, DF
2014

Este trabalho é dedicado a todas as pessoas que, assim como eu, acreditam que a Ciência é um instrumento transformador da sociedade e que, através dela, é possível romper barreiras ideológicas que limitam a mente humana.

Agradecimentos

Agradeço à toda minha família que, apesar de tantas diferenças, sempre me forneceu todas as condições necessárias para alcançar a posição onde hoje me encontro. Todo o apoio, paciência e compreensão foram imprescindíveis para percorrer o caminho até aqui.

Aos meus colegas da faculdade pela ajuda mútua nos estudos desde o início do curso e pelas boas risadas proporcionadas nos momentos de descontração.

Ao meu orientador Gilmar pelos bons conselhos e conhecimentos transmitidos.

E ao Tiago, meu coorientador e também chefe no meu atual ambiente de trabalho, pelo ótimo relacionamento e por me fazer adentrar neste imenso campo que é a modelagem e verificação de sistemas eletrônicos.

Tell me what you can hear
And then tell me what you see
Everybody has a different way
To view the world
I would like you to know
When you see the simple things
To appreciate this life
It's not too late to learn
(Iron Maiden - Different World)

Resumo

O presente Trabalho de Conclusão de Curso, propõe-se a desenvolver um ambiente de verificação funcional para modelos de processadores em nível transacional desenvolvidos em linguagem HDL (*Hardware Description Language*). Em especial, serão tratados modelos de processadores descritos em SystemC no padrão TLM-2.0 sem precisão temporal. Como estudo de caso, será utilizado o processador MIPS Plasma de 32 bits e cinco estágios de *pipeline* implementado pelo aluno Tiago Trindade da Silva, do programa de doutorado em Engenharia de Sistemas Eletrônicos e Automação do Departamento de Engenharia Elétrica, Universidade de Brasília. Na primeira parte do trabalho, foi feito um levantamento geral de metodologias praticadas no mercado que atendem os requisitos de verificação funcional necessários para validar modelos descritos em SystemC. Dentre várias metodologias e ferramentas encontradas, a UVM (*Universal Verification Methodology*) foi a escolhida para o desenvolvimento deste trabalho, pois atende por completo as necessidades de comunicação, interface e estrutura dos ambientes de verificação que deseja-se construir. Em conjunto com a UVM, utiliza-se a biblioteca UVM *Connect*, a qual possibilita a interação do código em SystemVerilog, proveniente do ambiente de verificação, com o código em SystemC, proveniente do modelo de processador testado. Nesta segunda e última etapa do trabalho, são mostradas as fases de planejamento, execução do processo de verificação funcional e resultados obtidos. A fase de planejamento é constituída pela definição das métricas de cobertura baseadas em técnicas de verificação de processadores e também pela elaboração do plano de verificação. A fase de execução consiste na criação de códigos que compõem ambientes de verificação que buscam exercitar o modelo de processador em seus aspectos funcionais. Diferentes tipos de testes aleatórios são gerados para alcançar pontos críticos que seriam dificilmente encontrados com simulações e testes comuns.

Palavras-chaves: verificação funcional. métricas de cobertura. metodologia. UVM. TLM. MIPS.

Abstract

This Final Paper proposes the development of a verification environment for processors implemented in HDL (Hardware Description Language) at the transaction level, specially, those described in SystemC TLM-2.0 standard, with un-timed code style. It will be used as design under verification the 32 bits MIPS processor with five pipeline stages implemented by the student Tiago Trindade da Silva of the doctoral program in Electronic Systems and Automation Engineering of the University of Brasília. In the first part of this work, it was made a research of methodologies used in the market which could meet the requisites of functional verification needed to validate a SystemC model. Among several methodologies and tools found, the UVM (Universal Verification Methodology) was chosen for the development of this work, because it meets all the requirements of communication, interface and architecture of the verification environment which is wished to develop. The usage of UVM with SystemC models require another library called UVM Connect, which connects the SystemVerilog code, from the testbench, with the C++ code, from the model. In this second and last step of the work, are presented the planning phases, project execution and results. The planning phase consists of defining the coverage metrics and the formulation of verification plan. The execution phase consists in write the code that composes the testbench environment. Different types of tests are executed for reach the corne cases of the project.

Key-words: functional verification. coverage metrics. methodology. UVM. TLM. MIPS.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
ASI	*Accellera System Initiative*
AT	*Approximated-Timed*
AVM	*Advanced Verification Methodology*
BFM	*Bus Functional Model*
CDV	*Coverage-Driven Verification*
CPU	*Central Processor Unit*
DUT	*Design Under Test*
EDA	*Electronic Design Automation*
*e*RM	*e Reuse Methodology*
ESL	*Electronic-System Level*
*e*VC	*e Verification Component*
FIFO	*First In, First Out*
FPGA	*Field-Programmable Gate Array*
GPR	*General Purpose Register*
HW	*Hardware*
HDL	*Hardware Description Language*
HLS	*High-Level Synthesis*
HVL	*Hardware Verification Language*
IP	*Intellectual Property*
ISA	*Instruction Set Architecture*
LT	*Loosely-Timed*
OO	Orientação a Objetos

OSCI	*Open SystemC Initiative*
OVM	*Open Verification Methodology*
OVM-ML	*OVM Mixed Languages*
PCA	*Pin and Cycle Accurate*
POA	Programação Orientada a Aspectos
RTL	*Register Transfer Logic*
RAM	*Random Access Memory*
ROM	*Read Only Memory*
SW	*Software*
SAM	*System Architecture Model*
SC	*SystemC*
SoC	*System-on-Chip*
TLM	*Transaction Level Modeling*
UART	*Universal Asynchronous Receiver/Transmiter*
ULA	Unidade Lógica Aritmética
URM	*Universal Reuse Methodology*
UT	*Un-Timed*
UVM	*Universal Verification Methodology*
VIP	*Verification Intellectual Property*
VHDL	*Very High Speed Integrated Circuits Hardware Description Language*
VLSI	*Very-Large-Scale integration*
VMM	*Verification Methodology Manual*

Sumário

1 Introdução

A evolução tecnológica trouxe com ela a possibilidade de criar circuitos integrados com trilhões de transistores que constituem sistemas de alta complexidade. Estes sistemas podem ser processadores de alto desempenho, sistemas multiprocessados, microcontroladores ou um sistema em *chip*, conhecido como SoC (do inglês *System-on-Chip*). O desenvolvimento destes tipos de sistemas tem sido um desafio para projetistas de todo o mundo, dada a dificuldade de se manter a conformidade entre o sistema desenvolvido e as especificações iniciais [1]. Segundo Sangiovanni-Vincentelli, McGeer e Saldanha [2], em pouco tempo será alcançado um ponto onde será impossível verificar a exatidão de um sistema sem introduzir uma metodologia de verificação no processo de design.

A complexidade dos grandes sistemas eletrônicos não vem apenas do contínuo aumento de funcionalidades do sistema, mas também da quantidade de requisitos rigorosos impostos sobre eles, tais como o prazo limite de entrega do produto para comercialização, segurança, confiabilidade e requisitos de performance.

A pressão no prazo de entrega do projeto, junto com a vasta quantidade de componentes que são necessários para alcançar desejadas funcionalidades, tornam impossível uma única companhia desenvolver e fabricar todo um sistema eletrônico dentro do tempo estimado e com custo acessível. Destarte, a reutilização de projetos, sejam eles blocos de um sistema eletrônico ou o próprio ambiente de verificação, e a compra de Propriedades Intelectuais (IP) se tornaram uma necessidade [3]. Entretanto, os IP *cores* também devem passar pelo processo de verificação antes de serem comercializados, garantindo, assim, segurança, confiabilidade e robustez ao usuário final [4].

1.1 Motivação e Justificativa

Neste cenário, houve uma mudança de paradigma, transferindo a maior parte dos esforços da equipe de engenheiros de *design* para a equipe de engenheiros de verificação [5]. Molina e Cadenas [4] cita uma estatística de que 60 a 70% de todo o ciclo de produção de um sistema lógico complexo é dedicado à tarefas de verificação. Assim, esta mudança de paradigma colocou a verificação no topo da linha de pesquisa e desenvolvimento da indústria de ferramentas de criação de circuitos integrados (EDA, *Electronic Design Automation*) [6].

O gargalo na verificação é um efeito do aumento do nível de abstração utilizado para descrever sistemas com alta complexidade. Como mostra a figura 1, a tecnologia de manufatura em silício cresce disparadamente se comparada com a capacidade de mode-

lagem e verificação. Observando este gráfico simplificado, notam-se duas lacunas geradas pela discrepância entre o avanço das tecnologias. A lacuna de verificação surge da dificuldade em gerar testes que verifiquem 100% de um modelo. Isto também é influenciado pela falta de metodologias eficazes, condizentes com o cenário de complexidade já exposto. Por conseguinte, a lacuna de produtividade é gerada pela falta de verificação. Em suma, não se pode produzir aquilo que não foi verificado e validado [4].

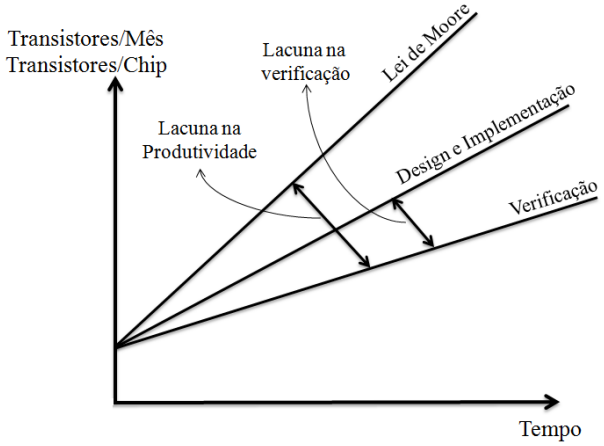

Figura 1: Lacuna no desenvolvimento do *design* e verificação [4].

Pagliarini [6] cita um estudo o qual mostra que o índice de sucesso na primeira rodada de manufatura em silício caiu de aproximadamente 50% para 35% num intervalo de quatro anos. Esta tendência é mostrada na figura 2.

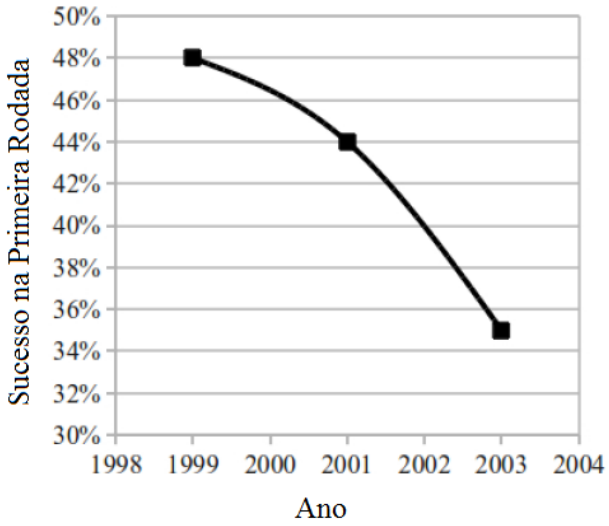

Figura 2: Decaimento no sucesso logo na primeira rodada de manufatura [6].

Pagliarini [6] também cita um estudo feito posteriormente, em 2007, pela Far West Research e a Mentor Graphics que identifica as fontes dos erros em projetos de *chips* (figura 3). Observa-se que os erros mais comuns são provenientes de funções lógicas, os quais são detectados justamente no processo de verificação funcional.

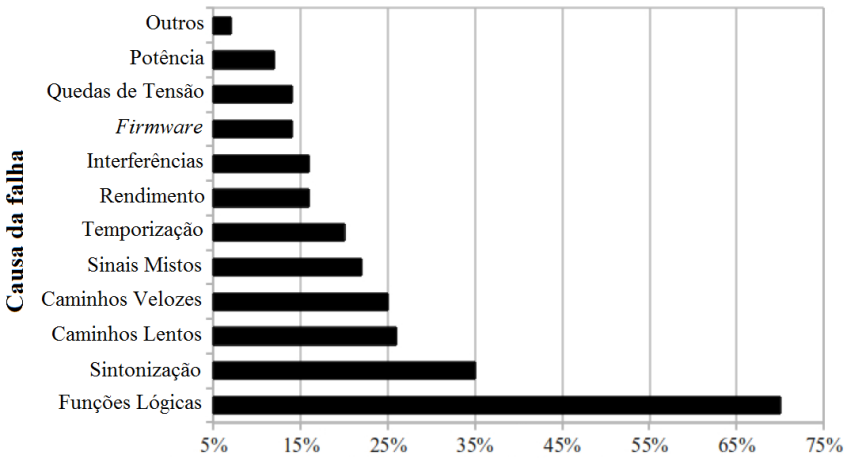

Figura 3: Percentual de falhas em projetos [6].

Estes fatos são os principais motivos para fomentar a pesquisa no campo das metodologias de verificação. Obter êxito logo na primeira rodada de projetos de sistemas eletrônicos não é mais uma opção. Portanto, é com este direcionamento que o presente trabalho irá se desdobrar.

1.2 Objetivos

O presente Trabalho de Conclusão de Curso tem caráter complementar ao programa de doutorado em Engenharia de Sistemas Eletrônicos e Automação do Departamento de Engenharia Elétrica, Universidade de Brasília, de Tiago Trindade da Silva. Seu trabalho está focado na modelagem transacional de núcleos de processamento para plataformas virtuais. Estes núcleos de processamento têm como base o ISA de um processador MIPS32 e estão sendo desenvolvidos em linguagem SystemC no padrão TLM-2.0 sem precisão temporal. Diversas versões já foram desenvolvidas e a versão utilizada para o desenvolvimento do presente trabalho é descrita com mais detalhes no capítulo de Estudo de Caso.

Como o doutorado de Tiago Trindade encontra-se em andamento, este trabalho tem caráter concorrente, de forma a verificar e validar uma das versões de processador utilizando métodos mais eficientes e dinâmicos, condizentes com o praticado atualmente no mercado. Desta forma, tem-se como objetivo principal a criação de um ambiente de verificação utilizando a metodologia UVM (*Universal Verification Methodology*). Deseja-se obter dados funcionais do modelo que confirmem a corretude da arquitetura através da geração de testes diretos e aleatórios, visando alcançar pontos críticos do modelo que seriam dificilmente encontrados em simulações e testes comuns. Para isto, um plano de verificação é elaborado com base no ferramental disponível e, sequencialmente, há a execução das etapas de verificação funcional, gerando resultados que mostram a presença

de erros ou falhas encontradas.

1.3 Estrutura do Documento

O corpo deste documento está dividido em oito capítulos. Os capítulos dois e três são dedicados a abordar os principais conceitos e aspectos que permeiam o mundo da verificação funcional, englobando as métricas de cobertura existentes e as etapas do processo de verificação funcional.

O capítulo subsequente mostra o estado da arte, descrevendo as principais características de diversas metodologias existentes no mercado. Também é exposto um quadro comparativo que aborda de forma objetiva as diferenças entre estas metodologias e seus pontos fortes e fracos. Uma ênfase maior é dada à UVM, visto que ela é a metodologia utilizada neste trabalho e necessita ser entendida mais a fundo.

O capítulo seguinte introduz o conceito de modelagem de sistemas, o qual traz a teoria por trás da abstração de detalhes no desenvolvimento de modelos em nível transacional. São abordados, principalmente, os conceitos de ESL (*Electronic System-Level*) e do padrão TLM-2.0 (*Transaction Level Modeling*) que possibilitam, através da biblioteca SystemC do C++, a modelagem de sistemas de alta complexidade. Este capítulo serve de norte para que o estudo de caso seja apresentado e seja possível conhecer sua arquitetura e funcionamento.

Então, já tendo conhecimento do modelo a ser verificado, o plano de verificação é exposto junto com a metodologia utilizada para construir os ambientes de verificação no sexto capítulo. Esta parte do texto retrata, com base no estudo de caso, aquilo que deseja-se alcançar com os testes de verificação funcional. Por fim, os resultados obtidos são discutidos, incluindo também comentários sobre os códigos elaborados e suas respectivas finalidades.

Todos os códigos gerados encontram-se em anexo a este documento.

2 Conceitos de Verificação Funcional

Este capítulo tem o intuito de introduzir o leitor aos primeiros conceitos necessários para o entendimento do processo de verificação, começando com o conceito de verificação e os tipos de verificação existentes. Em seguida, são retratadas as diferentes abordagens que podem ser utilizadas na verificação de um modelo e no que elas implicam. Por fim, as métricas de cobertura e sua importância na verificação funcional são abordadas.

2.1 O Que é Verificação?

A verificação é um processo utilizado para demonstrar o grau de corretude de um determinado projeto. Este processo afirma se o que está sendo verificado de fato implementa as funcionalidades descritas nos requisitos de projeto [7].

Um modelo, doravante denominado DUT (*Design Under Test*), pode ser verificado basicamente de duas formas: através de simulações ou de códigos que estimulam e checam o modelo automaticamente [8]. Simulações podem gerar gráficos e uma quantidade imensa de dados, os quais necessitam ser analisados posteriormente por um indivíduo (ou um grupo de indivíduos). Desta forma, a análise de dados de forma manual pode ser eficaz para modelos extremamente simples, porém, para sistemas com alguma complexidade intrínseca, este método não garante confiabilidade, pois torna-se difícil o manejo da grande quantidade de informação [7].

O processo de verificação através de códigos consiste em gerar um ambiente virtual que estimula o DUT através de sequências de sinais (ou de transações) pré-determinadas e monitoram as saídas para analisar o seu comportamento. Este conjunto de códigos constitui o ambiente de verificação e recebe o nome de *testbench*. Esta prática garante maior confiabilidade e eficiência, visto que os dados são gerados, colhidos e validados através de lógica computacional. Logo, tendo em vista estas características, o *testbench* deve modelar da melhor forma possível o ambiente no qual o DUT irá trabalhar quando implementado fisicamente [7]. Além disto, os códigos que constituem o ambiente de verificação podem ser escritos em diversas linguagens, desde o VHDL [9], passando pelo Verilog [10] e SystemVerilog [11], até o SystemC [12]. A figura 4 ilustra, de forma simples, como o *testbench* ele interage com o DUT.

O primeiro passo para iniciar uma verificação através de códigos é a determinação de dois pontos: o ponto de início e o ponto de reencontro [7]. Um ponto de reencontro corresponde à primeira instrução dinâmica em um programa onde pode-se esperar que os diversos caminhos do código retornem, independentemente do resultado ou do alvo do

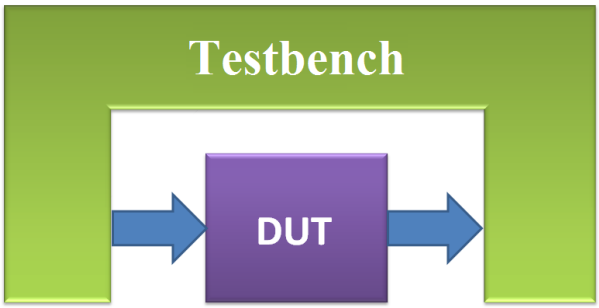

Figura 4: Modelo básico de um *testbench* para verificação

ramo em execução [13]. É importante saber onde estes pontos estão para poder entender quais transformações estão sendo verificadas.

A saber, existem dois tipos de verificação:

- Verificação formal
- Verificação funcional

Cada um dos tipos verifica aspectos diferentes, pois eles possuem pontos de início e de reencontro distintos.

A verificação formal, também referenciada como verificação estática, é ramificada em duas grandes categorias: checagem de equivalência e checagem de modelo. A checagem de equivalência consiste em um processo que prova matematicamente que a origem e a saída são logicamente equivalentes e que a transformação preservou suas funcionalidades (figura 5). Na maioria das vezes, este procedimento é utilizado para comparar se um modelo RTL (*Register Transfer Logic*) implementa corretamente determinada *netlist*.

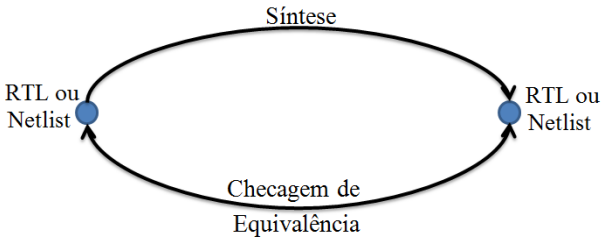

Figura 5: Caminhos da checagem de equivalência

Já a checagem de modelo procura por problemas gerais ou violações das regras definidas sobre o comportamento do projeto. Neste processo, as asserções ou características de um projeto são formalmente provadas ou negadas a partir das especificações e do modelo em HDL (*Hardware Description Language*). Os caminhos desta abordagem de verificação estão expostos na figura 6. Pode-se, por exemplo, encontrar estados isolados em um circuito sequencial ou pode-se verificar o comportamento das interfaces de um sistema [7].

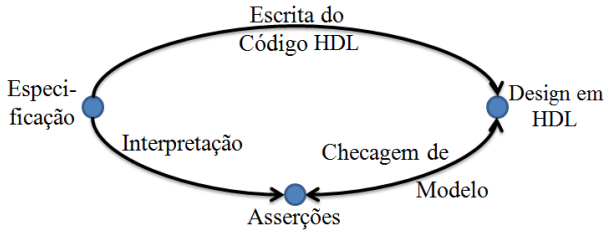

Figura 6: Caminhos da checagem de modelo

Já na verificação funcional, o principal propósito é assegurar que o projeto implementa as funcionalidades designadas nas etapas iniciais do projeto [7]. Esta etapa de transição do processo pode ser crítica, pois é preciso transpor toda a documentação escrita para um código em HDL, como mostra a figura 7, e isto torna o procedimento suscetível a falhas [4].

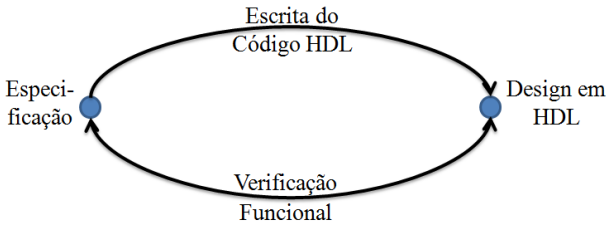

Figura 7: Caminhos da verificação funcional

É importante notar que, a menos que a especificação seja escrita em uma linguagem formal com semântica precisa, fica impossível provar que o projeto alcança os objetivos da especificação. A verificação funcional pode mostrar que o projeto atende às suas especificações, mas não pode provar isto. E ainda mais: é possível provar a presença de erros e falhas, mas não é possível provar a sua ausência [7].

Neste trabalho, o foco é exclusivamente a verificação funcional, que é o processo chave para demonstrar que o modelo de processador desenvolvido em HDL implementa as funcionalidades desejadas, tais como acesso à memória, escrita e leitura em registradores, tratamento de interrupções, desvios de programa, etc. Na seção seguinte, serão apresentados os três tipos de abordagens com as quais pode-se trabalhar em um processo de verificação funcional.

2.2 Abordagens da Verificação Funcional

A verificação funcional pode ser realizada utilizando três abordagens complementares, porém diferentes: *black-box*, *white-box* e *grey-box*. O tipo de abordagem implica no nível de abstração e complexidade com o qual o ambiente de verificação será construído e cada situação irá caracterizar um tipo de abordagem diferente. Dependendo da fase do projeto na qual está sendo feita a verificação, um tipo de abordagem será mais adequado

que os demais. A decisão da abordagem a ser utilizada deve ser determinada no plano de verificação de acordo com a etapa ou especificações do projeto. Segundo Bergeron [7], Rodrigues [14] e Vasudevan [15], as características de cada tipo serão apresentadas a seguir.

2.2.1 Abordagem *Black-box*

Na abordagem *black-box*, a verificação funcional é feita sem nenhum conhecimento sobre a estrutura interna do DUT. A única ligação com o DUT é feita através de suas interfaces e não se deve acessar nenhum componente ou bloco interno do modelo. Isto, obviamente, provoca uma falta de visibilidade e de controlabilidade, o que dificulta observar a resposta da entrada e localizar a fonte do problema. Esta dificuldade é acarretada pela demora entre a ocorrência do problema e a aparição de seus sintomas.

As vantagens desta abordagem é que ela não depende de uma implementação específica. A verificação pode ser realizada e mostrar que um modelo particular implementa determinadas funcionalidades, independentemente da implementação. Com isto, consegue-se escrever *testbenches* reutilizáveis, ou seja, que podem ser utilizados na verificação de outros dispositivos. Além disto, esta abordagem é a única que permite ser utilizada em paralelo com a fase de implementação do projeto porque não é necessário saber de antemão nenhum detalhe da construção interna. Outra possibilidade de utilização desta abordagem é para a criação de *testbenches* genéricos que servirão para verificar vários modelos com uma determinada funcionalidade, porém com implementações diferentes. Por exemplo, barramentos AMBA (*Advanced Microcontroller Bus Architecture*) implementados em VHDL e Verilog.

2.2.2 Abordagem *White-box*

Esta abordagem possui alta visibilidade e controlabilidade sobre as partes internas do DUT. Este método possui a vantagem de permitir a criação, com facilidade, de cenários e estímulos para as entradas do modelo e isolar determinadas funções do mesmo. Isto possibilita encontrar erros com maior rapidez.

Todavia, este tipo de abordagem está altamente acoplado a detalhes de implementação do DUT e não pode ser reutilizado com facilidade em projetos futuros. Alterações no DUT podem resultar em modificações no ambiente de verificação. Isto também requer conhecimento refinado sobre a implementação do DUT para saber quais condições de teste devem ser gerada e quais resultados devem ser observados. Esta abordagem é adequada quando se verifica blocos separados de um modelo com o intuito de isolar defeitos com maior precisão.

2.2.3 Abordagem *Grey-box*

A abordagem *grey-box* contém elementos das duas abordagens anteriores: a abstração da *black-box* e a dependência da implementação da *white-box*. A primeira pode não exercitar totalmente partes do projeto e a segunda não é portável. Assim, esta abordagem constitui uma mistura das duas abordagens anteriores.

Por exemplo, esta abordagem pode ser utilizada para aumentar as métricas de cobertura. Estímulos podem ser especificamente gerados para determinadas linhas de código ou para verificar funcionalidades específicas.

2.3 Métricas de Cobertura

As métricas de cobertura são ferramentas essenciais para diretores de projeto. Baseando-se nelas, pode-se obter uma fotografia da situação atual do projeto e pode-se avaliar e medir o progresso na etapa de desenvolvimento [7].

Quando se encontram erros durante a simulação de um projeto, a tendência é exercitar mais e mais o DUT até que não apareçam mais falhas. Esta abordagem é boa, contudo, para projetos de alta complexidade, é complicado saber quando parar os testes. Para a grande maioria dos projetos atuais, não é possível fazer com que todos os cenários possíveis sejam executados em tempo aceitável. Para contornar este problema, emprega-se a técnica de cobertura para detectar se o projeto já foi suficientemente exercitado durante a verificação funcional [14].

A seguir, serão introduzidos os dois principais tipos de cobertura existentes: cobertura de código e cobertura funcional [8]. Posteriormente, será introduzido o conceito da verificação dirigida por cobertura (CDV, do inglês *Coverage-Driven Verification*), a qual engloba conceitos de cobertura de código e funcional.

2.3.1 Cobertura de Código

A cobertura de código define um espaço de cobertura de implementação implícita. Um espaço de cobertura de implementação implícita é tal que as métricas de cobertura são definidas pela fonte que está sendo observada e extraídas da implementação do dispositivo [8].

Para esta abordagem, a ferramenta de análise de cobertura vai reportar quais partes do código foram ou não executadas durante a simulação, além de marcar a quantidade de vezes. Este tipo de cobertura é importante por mostrar se alguma parte do projeto não foi exercitada, já que a existência de partes do código não executadas durante a simulação pode ser um ponto indicador de erros na implementação [14].

A cobertura de código é um tipo de métrica estrutural e pode ser de vários tipos. De acordo com Piziali [8], os tipos mais comuns são a cobertura de linhas, cobertura de declarações, cobertura de ramos, cobertura de condições, cobertura de eventos e a cobertura de mudanças, as quais são explicadas brevemente a seguir:

- **Cobertura de Linhas**: esta métrica relata quais linhas de código do DUT foram ou não foram executadas durante a simulação. A cobertura completa de linhas é atingida quando todas as linhas executáveis são executadas acima de um valor limiar de vezes. Esta quantidade limiar é normalmente um argumento definido pelo usuário.

- **Cobertura de Declarações**: esta métrica relata quais declarações do código de descrição de hardware foram ou não executadas. Esta métrica é mais precisa do que a cobertura por linhas porque declarações podem ocupar múltiplas linhas e mais de uma declaração pode estar contida em uma única linha.

- **Cobertura de Ramos**: a métrica de cobertura de ramos mede nas estruturas de controle de fluxo quais ramos são utilizados. Com esta abordagem, é possível identificar se, durante a simulação, todos os ramos de execução possíveis junto às estruturas condicionais do código são contempladas. Este modelo de cobertura serve para indicar se condições fora do caminho normal são observadas na verificação.

- **Cobertura de Condições**: esta métrica grava o número de vezes que cada expressão Booleana das estruturas condicionais gera valores verdadeiros ou falsos (1 ou 0, respectivamente). Ela considera todas as possibilidades existentes para expressões dependentes de mais de uma variável.

- **Cobertura de Eventos**: a métrica de cobertura de eventos conta o número de vezes que um evento é acionado, ocorre ou é emitido. Em SystemVerilog, um evento é acionado quando o valor de um registrador ou fio muda. Em VHDL, um evento ocorre no sinal quando o valor deste sinal muda.

- **Cobertura de Mudanças**: este tipo de métrica relata o número de vezes que cada bit de um registrador ou fio do DUT muda seu valor. Com ele, visa-se buscar problemas de controlabilidade em determinadas variáveis do sistema.

2.3.2 Cobertura Funcional

Antes de explicar o que é a cobertura funcional, é importante entender por que a cobertura de código não é o suficiente. Como mencionado anteriormente, a cobertura de código irá refletir em porcentagem o quanto de um código é exercitado. As ferramentas

de cobertura de código mapeiam a execução do código instrumentando ou mesmo modificando o código HDL. Contudo, o motivo para esta métrica não ser suficiente é simples: a maioria dos requisitos de cobertura funcional não podem ser mapeados através das estruturas do código.

A cobertura de código não olha para sequências de eventos, como o que aconteceu antes, durante e depois de uma dada linha de código ser executada. Em tese, sempre que é necessário relacionar eventos e cenários, a cobertura de código falha. Mesmo assim, a cobertura de código ainda é necessária, pois é um tanto inaceitável desenvolver um código "morto" ou que não pode ser verificado.

Por outro lado, a verificação funcional observa o projeto do ponto de vista do usuário. São exemplos de questões que a verificação funcional procura arduamente responder [6]:

- Todos os cenários típicos foram cobertos?
- Foram testados cenários que geram erros?
- Quais são os casos críticos do projeto?
- Como levar em consideração cenários ou sequências de protocolos?

Esta métrica define um espaço de cobertura de implementação explícita. O propósito de se medir a cobertura funcional é medir o progresso da verificação pela perspectiva dos requisitos funcionais do dispositivo. Os requisitos funcionais são impostos tanto nas entradas quanto nas saídas do dispositivo (e na relação que há entre elas), baseados nas especificações do modelo.

Desde que o comportamento completo do dispositivo é definido por suas entradas e saídas, o espaço de cobertura funcional que captura esses requisitos é referido como modelo de cobertura. A fidelidade do modelo de cobertura diz o grau com o qual o modelo captura os requisitos do projeto. Se está sendo modelado um controlador de registrador com 18 valores especificados e o modelo de cobertura define todos os 18 valores, então este é um modelo de alta fidelidade. Já se o projeto consistir em um barramento de 2^{32} valores especificados e o modelo de cobertura abrange somente 2^{16} valores, uma lacuna substancial é introduzida. Consequentemente, este será um modelo de baixa fidelidade [8];

A cobertura funcional, portanto, depende do domínio da aplicação. Isto significa que a especificação dos critérios de cobertura não podem ser extraídos diretamente do código em HDL. Ela deve ser feita pelo engenheiro de verificação. Os critérios normalmente utilizados para cobertura funcional são: cobertura de valores escalares individuais e cobertura cruzada.

Na cobertura de valores escalares, o engenheiro especifica o conjunto de valores relevantes que devem ser observados na verificação, seja como estímulos na entrada ou como resultados na saída. E na cobertura cruzada, ele mede a combinação de diversos valores. A implementação da cobertura cruzada segue o mesmo princípio da cobertura de valores escalares. A diferença é que na cobertura cruzada vários valores são coletados ao mesmo tempo [14].

2.3.3 Verificação Dirigida Por Cobertura

A verificação dirigida por cobertura (CDV, do inglês *Coverage-Driven Verification*) é uma metodologia na qual o plano de verificação precede todo o resto do processo de verificação [8]. Como será visto no próximo capítulo, montar um plano de cobertura significa definir uma estratégia para medir o progresso da verificação e saber quando a verificação chega ao fim.

A CDV combina a geração automática de testes, *testebenches* com auto-checagem e métricas de cobertura para reduzir substancialmente o tempo gasto verificando um modelo. Segundo Accellera [16], o propósito da CDV é:

- Eliminar o esforço e o tempo gasto ao criar-se centenas de testes;
- Assegurar uma verificação completa estabelecendo metas progressivas;
- Receber notificações antecipadas de erros e implantar a verificação em tempo de execução para simplificar a depuração.

O fluxo da CDV é diferente daquele utilizado nos testes diretos. Primeiramente, deve-se estabelecer as metas de verificação através de um processo organizado de planejamento. Posteriormente, cria-se um *testbench* dinâmico que gera estímulos válidos e envia-os para o DUT. Monitores de cobertura são adicionados ao ambiente de verificação para medir o progresso e identificar funcionalidades não exercitadas. Verificadores são instalados para identificar comportamentos indesejados do DUT. As simulações são então iniciadas e a verificação pode ser realizada.

Utilizando a CDV, pode-se verificar completamente um projeto alterando-se os parâmetros do *testbench* ou trocando-se a semente para geração de estímulos aleatórios. Além disto, pode-se adicionar restrições aos valores gerados para alcançar as metas mais rapidamente. Ademais, há ainda a opção de mesclar estímulos aleatórios com estímulos diretos para obter a máxima cobertura possível.

Usando uma abstração maior (sem se prender a detalhes de implementação) e uma abordagem baseada nas funcionalidades, consegue-se obter um plano de verificação mais

legível, adaptável e reutilizável. Unindo todas estas características, a CDV possibilita que as fontes de cobertura sejam planejadas, observadas, classificadas e relatadas [16].

3 O Processo de Verificação

A literatura não entra em um consenso perfeito quanto às etapas do processo de verificação. Diferentes autores descrevem o fluxo de verificação de maneiras distintas, porém muito semelhantes em seus fundamentos. Neste contexto, escolheu-se a estrutura descrita por Piziali [8] que consiste em quatro etapas: plano de verificação, implementação do ambiente de verificação, *bring-up* e regressão. Dentre os modelos encontrados na literatura, este é o mais abrangente e a ele serão incorporados conceitos apresentados em outras obras que complementem suas etapas.

3.1 Plano de Verificação

A elaboração do plano de verificação é a primeira etapa do processo de verificação e serve para definir o que, obrigatoriamente, necessita ser verificado e como isto será verificado. Ele é utilizado para descrever o escopo do problema de verificação para o dispositivo, os objetivos da verificação, como o projeto será verificado e quais *testbenches* serão escritos [8].

É necessário determinar com o maior grau de segurança possível quando a verificação estará completa. A equipe de verificação, em conjunto com a equipe de projeto, necessita estabelecer quais são as características essenciais a serem exercitadas na verificação, sob quais condições testá-las e quais respostas esperar. Na documentação do plano de verificação, algumas características são prioridades e outras são opcionais [17]. Uma vez escrito o plano de verificação, sabe-se quantos *testbenches* serão escritos e qual a complexidade de cada um. O roteiro de verificação deve tentar paralelizar ao máximo os testes de forma a se ganhar tempo [8].

Vasudevan [15] enfatiza algumas informações que devem ser coletadas antes de escrever o plano de verificação:

- Especificações arquiteturais;
- Diferentes modos de operação do dispositivo;
- Tipos de comportamento apresentados pelo dispositivo em caso de funcionamento normal ou errôneo;
- Qualquer tipo de padrão que o dispositivo deva atender;
- Uma lista de entradas e saídas do modelo;

- Uma lista de cenários nos quais o modelo possa, provavelmente, apresentar funcionamento indevido;
- Aplicações onde o dispositivo será usado.

A verificação funcional pode ser decomposta em quatro grandes aspectos que irão compor a estratégia do processo de verificação [7] [18] [8]:

1. Nível de verificação
2. Medidas de cobertura
3. Geração de estímulos
4. Checagem de respostas

Cada aspecto define características importantes que irão compor o plano de verificação. Detalhá-los com afinco tornará o plano de verificação mais completo e, consequentemente, fará o processo de verificação abranger maior parte das funcionalidades do modelo a ser testado. A seguir, cada aspecto é explorado com maior ênfase.

3.1.1 Níveis de Verificação

Quando se está planejando a verificação, a primeira questão que deve surgir é a determinação do nível de detalhamento para o trabalho. Um modelo é potencialmente composto por vários níveis. Os engenheiros de *design* dividem o sistema em várias unidades lógicas e esta prática é conhecida como *design* hierárquico. Esta prática subdivide um problema complexo em partes mais manejáveis e possibilita aos engenheiros combinarem estas partes para formar blocos maiores [7].

Devido ao sistema ser dividido nestas unidades menores, a verificação toma vantagem das mesmas fronteiras hierárquicas. Podem existir diversos níveis de verificação. Aqui serão apresentados os níveis mais comuns, contudo um projeto pode contemplar mais ou menos níveis dependendo de sua complexidade [18]. A figura 8 exemplifica como estes níveis de hierarquia podem ser divididos em um projeto, partindo do nível de sistema (nível mais alto) até o nível de projetista (nível mais baixo).

O nível de projetista é o mais baixo e recebe este nome porque, normalmente, é verificado pelo próprio projetista. Como um projeto de grande porte possui muito blocos funcionais, não é factível que um engenheiro de verificação teste todos os blocos. Contudo, os blocos com maior risco de falhas devem ser testados.

Na maioria dos projetos complexos, a verificação no nível de unidade se faz necessária. Neste nível, as interfaces e funções tendem a ser mais estáveis e bem definidas do

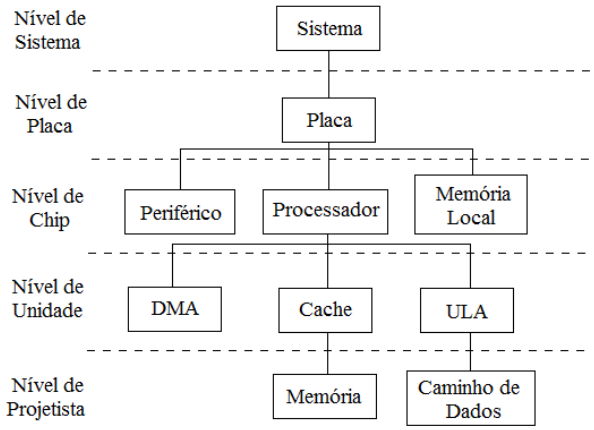

Figura 8: Níveis de Verificação [18]

que no nível mais baixo, porque uma unidade tende a ter especificações mais formais de funcionamento. Com isto, a equipe de verificação pode criar *testbenches* mais avançados, usando estímulos aleatórios e auto-checagem.

O nível de *chip* é composto por várias unidades e possui fronteiras muito bem definidas. O intuito desta verificação é saber se as unidades foram corretamente conectadas e se o modelo atende a todos protocolos de comunicação entre as unidades.

Uma placa é um aglomerado de *chips* conectados através de alguma lógica. O propósito da verificação neste nível é conferir a conexão entre os *chips*, a integração e o *layout* da placa.

A definição de sistema é diferente entre os diversos segmentos da indústria. Em alguns casos um *chip* pode ser um sistema (SoC); em outros casos, um sistema pode ser um enorme servidor composto por centenas de *chips*. Porém, no contexto da verificação, a definição básica de um sistema é uma partição lógica composta por componentes independentemente verificados. O foco da verificação aqui é testar a interação de todos os componentes e de funções particulares que garantem o comportamento desejado do sistema [18].

O conceito de nível de verificação ainda se entrelaça fortemente com a abordagem da verificação funcional descritos na seção 2.2. Níveis mais baixos de verificação tendem a utilizar a abordagem *white-box* com o intuito de exercitar fortemente cada funcionalidade do módulo. Como o nível de verificação é mais baixo, isto acaba requerendo um conhecimento maior da arquitetura interna do bloco a ser testado. Já em níveis de verificação mais altos, a tendência é de verificar o modelo segundo a abordagem *black-box*, pois é difícil aprofundar-se em detalhes dos submódulos do sistema. Assim, isto requer um nível de abstração maior.

Então, estando definido o nível de verificação a ser trabalhado de acordo com a

necessidade do projeto, pode-se partir para a determinação dos aspectos de verificação, abordados a seguir.

3.1.2 Medidas de Cobertura

A seção de medidas de cobertura de um plano de verificação deve descrever a dimensão do problema de verificação e como ele está dividido. Deve-se delegar responsabilidades para medir o progresso em meio aos tipos de cobertura descritos no capítulo anterior (*e. g.* cobertura funcional e cobertura de código). Esta parte do documento deve conter informações detalhadas de cada modelo de cobertura a ser adotado e que estejam em conformidade com as informações coletadas anteriormente.

3.1.3 Geração de Estímulos

A responsabilidade dos estímulos é exercitar completamente o modelo, *i. e.* fazer o mesmo exibir todos os possíveis comportamentos. A geração dos estímulos pode seguir uma lógica sequencial ou aleatória, bem como pode conter valores válidos ou inválidos [8] [7] [15].

Os estímulos válidos estão cercados por regras de dados e de tempo. Por exemplo, ao gerar uma instrução para um processador, o *opcode* deve conter uma instrução válida do conjunto de instruções de sua respectiva arquitetura. E no caso de estar gerando pedidos de pacotes de dados em uma linha de transmissão, deve-se obedecer a temporização do protocolo que está implementado no barramento.

Uma estratégia frequentemente utilizada no nível de sistema é a verificação aleatória. A verificação aleatória não quer dizer que aplica-se aleatoriamente zeros e uns nas entradas do projeto. Na verdade, as entradas são submetidas a valores válidos, porém, é a sequência dessas operações e o conteúdo do dado transferido que é aleatório. A verificação aleatória serve para criar cenários de estímulos que não foram pensados durante a etapa de escrita do plano de verificação. Ela cria situações inesperadas e consegue atingir casos críticos que geram algum tipo de exceção.

Devido à sua natureza, os estímulos aleatórios são difíceis de especificar. A sequência de dados e operações deve ser uma representação fiel das condições de operação na qual o projeto irá trabalhar. Mais complicado ainda é prever a saída, pois não se sabe qual estímulo será gerado. Logo, deve-se estabelecer como respostas inválidas serão detectadas.

Contudo, há circunstâncias em que é desejável aplicar estímulos inválidos no DUT como, por exemplo, quando se está verificando uma lógica de detecção de erro.

3.1.4 Checagem de Respostas

Decidir como aplicar os estímulos é relativamente fácil. O engenheiro de verificação tem completo controle sobre a temporização e o conteúdo do estímulo. A etapa difícil é verificar a resposta. É preciso planejar como se determinará a resposta esperada e, depois, verificar se o modelo concedeu tal resposta. Em alguns casos, é difícil para o *testbench* verificar uma resposta que poderia ser imediatamente reconhecida como certa ou errada por um humano. Por exemplo, um *testbench* pode, com mais facilidade, checar os *pixels* de uma imagem, porém, um humano pode reconhecer de prontidão um círculo pintado de vermelho. Então, a estratégia de verificação deve levar em consideração estes casos de testes para automatizar ao máximo o processo. Para isto, existem duas estratégias empregadas: modelo de referência ou checagem distribuída.

A primeira estratégia consiste em produzir um conjunto de saídas que possa, posteriormente, ser comparado a um conjunto de resultados de referência de um *golden model*. O modelo de referência deve conter um certo nível de abstração para que a verificação funcional possa ser feita e não é recomendado que seja utilizado para verificar a implementação, caso contrário os custos seriam substancialmente altos, pois o modelo deveria seguir continuamente as alterações nas especificações de projeto.

Contudo, uma consideração a ser feita é que o modelo de referência também necessita ser verificado. Se um modelo de confiança não for obtido, isto ocasionará um processo recursivo.

Entretanto, é mais eficiente detectar problemas o mais cedo possível. Quando a resposta é checada durante o tempo de simulação, o erro é identificado e consertado com mais facilidade. Deste ponto de vista, a checagem distribuída usa monitores para capturar o comportamento do dispositivo. Este comportamento é então comparado a um valor esperado que pode ser obtido através da função de transferência do DUT [19].

Um marcador pode ser utilizado na checagem distribuída e consiste em uma estrutura de dados usada para armazenar tanto os valores esperados como os dados de entrada do dispositivo. A ligação do marcador no ambiente de verificação é mostrada na figura 9.

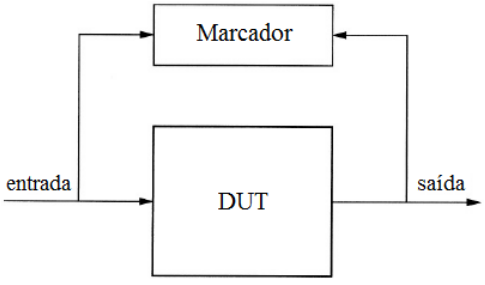

Figura 9: Interligação de um marcador no ambiente de verificação [8]

O DUT captura os estímulos, processa-os e escreve o resultado nas saídas. Cada

vez que o DUT escreve um resultado na saída, o respectivo dado de entrada é lido no marcador, transformado de acordo com as especificações e comparado com o resultado na saída. O marcador, por sua vez, sinaliza se houve um acerto ou um erro entre os dados comparados.

Normalmente, ambientes de verificação heterogêneos são escritos empregando ambas as abordagens. Por exemplo, um modelo de referência pode ser utilizado para testar o conjunto de instruções (ISA, do inglês *Instruction Set Architecture*) de um processador, porém a checagem distribuída é necessária para monitorar o protocolo de barramento [8].

Uma vez escrito o plano de verificação, a equipe de verificação pode colocá-lo em prática. A próxima seção aborda alguns pontos sobre a implementação do ambiente de verificação.

3.2 Implementação do Ambiente de Verificação

Uma vez que o plano de verificação está escrito (ou pelo menos o primeiro rascunho), ele servirá de especificação funcional para o ambiente de verificação. Deve-se pensar antecipadamente que a implementação do ambiente de verificação e de seus componentes enseja a reutilização. Isto é, construir novos componentes que sejam reconfiguráveis e que possam ser reutilizados posteriormente ou, simplesmente, reutilizar componentes disponíveis em uma biblioteca pessoal [20].

É por esta razão que a utilização de IPs tem se tornado um ponto crítico para a conclusão de projetos dentro do prazo e, da mesma forma, a verificação de IPs reutilizáveis também tem se tornado importante. Isto traz duas diretrizes: primeiro, sempre que possível, deve-se adquirir componentes de verificação em vez de construí-los. Segundo, sempre que for necessário escrever o próprio ambiente de verificação, deve-se fazê-lo o mais reutilizável possível [8].

A primeira tarefa pela qual o ambiente de verificação se responsabiliza é auxiliar o DUT nos seus primeiros passos de vida. Isto é conhecido como *bring-up* e será abordado na próxima seção.

3.3 *Bring-up*

O propósito da fase de *bring-up*[1] é eliminar todos os maus comportamentos e erros que podem vir a impedir o dispositivo de funcionar. Apesar do engenheiro de projeto

[1] O termo *bring-up* não possui uma tradução exata para a língua portuguesa. Ele se refere, dentre outras coisas, a auxiliar uma criança em seus primeiros passos de vida e a formar sua personalidade durante a fase de crescimento. Tendo em mente isto, uma analogia pode ser feita com a etapa de verificação de um projeto recém implementado.

normalmente verificar a funcionalidade básica do dispositivo durante sua fase de implementação, inevitavelmente, quando o projeto passa para a fase de verificação, muitos erros aparecem. A fim de facilitar a transição da fase de implementação para a fase de verificação, o engenheiro de verificação prepara um conjunto de simulações para demonstrar as funcionalidades básicas [8].

Estas simulações necessitam exercitar de uma forma muito restrita determinado comportamento do dispositivo. O motivo da restrição é facilitar o diagnóstico e a correção dos erros encontrados. Na fase de *bring-up*, uma hipótese é feita e deve-se provar que ela é verdadeira. Então, esta hipótese pode servir de base para construir uma outra hipótese mais complexa.

Uma vez que o dispositivo consegue rodar todas as simulações atingindo a cobertura completa, é necessário rodar novas simulações enquanto mudanças são feitas no modelo até que ele passe para a fase de manufatura.

3.4 Regressão

A definição no dicionário Michaelis [21] para regressão é "retornar a um estado anterior ou menos aperfeiçoado". Por isso, o propósito da etapa de regressão é detectar a reintrodução de erros que levem o projeto a "um estado menos aperfeiçoado". Isto consiste em continuar reexecutando os testes definidos no plano de verificação. Este é um passo necessário no ciclo de verificação por duas razões: a primeira se deve ao fato dos ambientes de verificação gerarem estímulos aleatórios, os quais formam diferentes cenários cada vez que se executa o teste; a segunda razão é que os testes devem ser rodados após serem feitos os ajustes no modelo para confirmar a correção [18] [15]. Alguns erros são facilmente reintroduzidos no projeto e necessitam ser encontrados rapidamente. Para isto, existem duas abordagens: regressão clássica e regressão autônoma [8].

A regressão clássica baseia-se em testes diretos que verificam uma determinada característica ou função do projeto. O critério para adicionar um teste ao conjunto de testes de regressão é:

- se ele verifica um comportamento fundamental
- se exercita bastante o projeto em poucos ciclos
- se já descobriu um ou mais erros no passado

Já a regressão autônoma é realizada por um ambiente de verificação autônomo segundo os aspectos de cobertura, geração de estímulos e checagem de resposta. Ela é preferida em relação à regressão clássica por fazer uso de ambientes de verificação dinâmicos e ser totalmente dirigida por cobertura.

Tendo concluído todas as etapas do processo de verificação, um modelo pode passar para a fase seguinte de síntese e, posteriormente, de manufatura, concluindo o ciclo de projeto de um sistema eletrônico.

4 Estado da Arte

Nos últimos anos diferentes metodologias de verificação foram utilizadas pelas equipes de verificação. A necessidade de utilização de uma metodologia de verificação se tornou clara nos dias atuais: conciliar um conjunto de padrões fornecidos pelas metodologias permite reutilizar certos aspectos da verificação, diminuindo, assim, o esforço total [6].

Este capítulo será dedicado a explorar as metodologias de verificação existentes e praticadas no mercado atualmente. Serão abordados aspectos básicos sobre o campo de aplicação, estrutura dos ambientes de verificação e pontos positivos e negativos da adoção de cada metodologia. Porém, antes de introduzir cada metodologia e discutir suas características, é interessante avaliar quais metodologias estão sendo mais utilizadas. Uma pesquisa de mercado feita por um consultor da área de EDA [22] mostrou informações relevantes sobre o assunto. O resultado pode ser visto na figura 10.

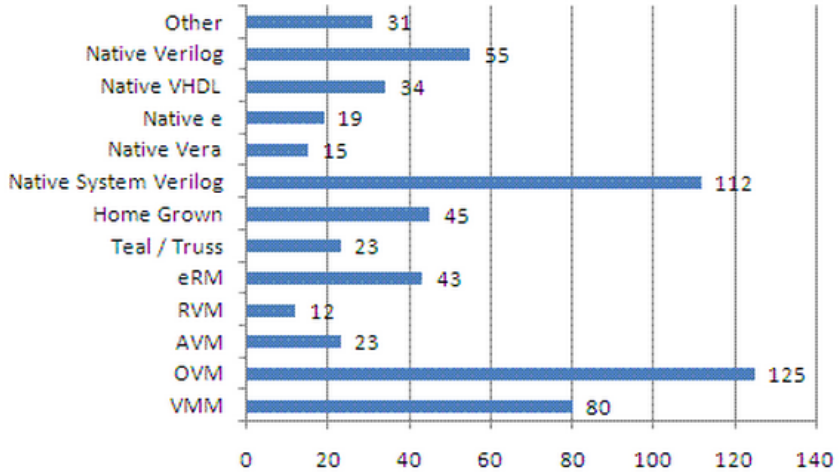

Figura 10: Práticas de verificação mais utilizadas, separadas por linguagem de programação ou metodologia [22]

Esta pesquisa reuniu dados coletados de 249 companhias ao redor do mundo que trabalham com o desenvolvimento de ASICs (*Application Specific Integrated Circuit*). O total excede o número de respondentes, pois eles podem utilizar mais de uma metodologia. Os resultados mostram que uma considerável parte dos respondentes não utiliza nenhuma metodologia de verificação, consequentemente, realizando a verificação através de uma linguagem de verificação nativa (*e. g.* SystemVerilog) ou de soluções desenvolvidas por eles mesmos. Ainda é possível visualizar o uso de linguagens que não são específicas para verificação, tais como o Verilog e o VHDL.

É importante notar que, algumas vezes, uma metodologia de verificação é especí-

fica de uma companhia de soluções EDA. Isto significa que apenas alguns simuladores são capazes de manusear tal metodologia. Atualmente, cada grande vendedor tem pelo menos um simulador em seu portfólio de soluções e, ligado eles, há pelo menos uma metodologia recomendada. Este é o caso da OVM suportada pela Cadence e da VMM suportada pela Synopsys. Uma quebra de paradigma ocorreu com a introdução da UVM, que é suportada pelo grupo Accellera e tem o propósito de ser uma metodologia de fato universal, suportada por diversos simuladores e adotada por diversas companhias de desenvolvimento de sistemas eletrônios [6] [23]. A evolução de todas estas metodologias é ilustrada na figura 11.

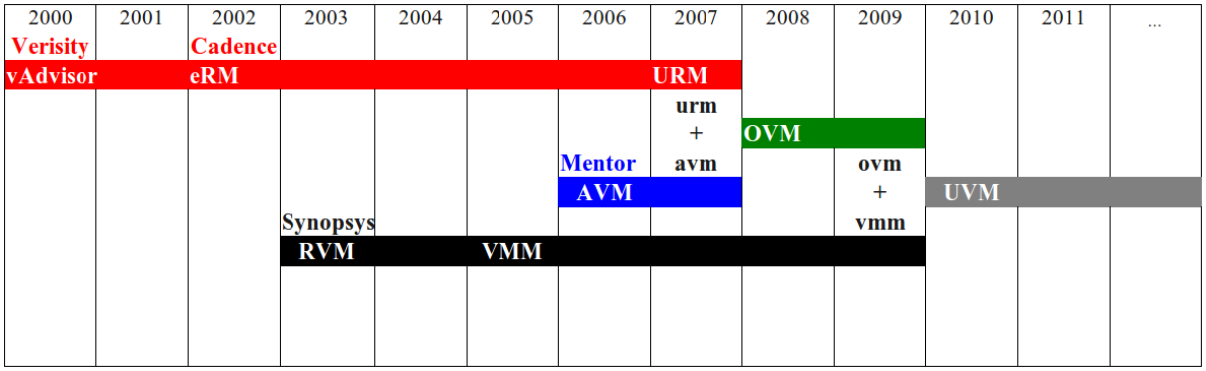

Figura 11: Linha do tempo que retrata a evolução e junção das metodologias de verificação [6] [23]

Nota-se que, ao longo do tempo, diferentes metodologias foram mescladas entre si, dando origem a novas metodologias mais robustas e condizentes com sua alocação no tempo. As metodologias que naturalmente evoluíram para novos padrões não serão abordadas neste capítulo, tendo em vista que a proposta do trabalho é a aplicação de uma metodologia atual e de amplo uso na indústria. Entende-se que, em uma faixa de pelo menos dez anos, estas metodologias não são mais capazes de acompanhar a realidade do mercado e da capacidade de projeto e manufatura existentes atualmente. Portanto, em consonância com a pesquisa de Gries [22], serão abordadas somente as metodologias *e*RM, VMM, OVM e UVM.

4.1 *e Reuse Methodology* (*e*RM)

A linguagem *e* é uma HVL (*Hardware Verification Language*) desenvolvida pela Verisity Design (atualmente incorporada à Cadence Design Systems). Quando a linguagem *e* é utilizada para implementar um modelo, a metodologia *e*RM deve ser seguida. A *e*RM está totalmente voltada para a reutilização do código escrito em *e* no desenvolvimento do ambiente de verificação. A *e*RM assegura a reutilização por fornecer os melhores métodos conhecidos para projeto, criação de códigos e empacotamento de códigos em *e* como componentes reutilizáveis [24].

Esta metodologia acompanha uma lista de novos conceitos introduzidos pela sintaxe e semântica da linguagem *e*. A característica mais marcante da *e* é sua semântica voltada para a Programação Orientada a Aspectos (POA), a qual inclui a semântica regular da Orientação a Objetos (OO) mais um grande nível de customização. Ao mesmo tempo que a POA faz a linguagem *e* única, ela a torna mais complexa do que uma linguagem normal do tipo OO [6].

A *e*RM criou o conceito de *e*VC (*e Verification Component*), o qual é um ambiente de verificação configurável e pronto para ser utilizado, focado em uma arquitetura ou protocolo específico, *e. g.* PCI Express, Ethernet ou USB. Cada *e*VC consiste em um conjunto completo de elementos para estimular, checar e coletar informação de cobertura da arquitetura ou protocolo ao qual se refere. Eles podem trabalhar com DUTs escritos em linguagem Verilog e VHDL e todos os simuladores suportados pela Specman [24].

No mais, esta metodologia dá suporte a ambientes de verificação que geram estímulos aleatórios e que aplicam a CDV. A metodologia também possibilita caminhos para a escrita de blocos na visão arquitetural (*i. e.* monitores, geradores de estímulos, *checkers*, etc.) [6]. A figura 12 mostra um típico *e*VC conectado a um DUT e servirá de exemplo para demonstrar o princípio desta metodologia.

Deve haver um *e*VC conectado a cada porta de comunicação, assim, embora não mostrado na figura 12, o *testbench* deve instanciar um outro componente específico para barramentos lógicos que faça o controle da respectiva interface.

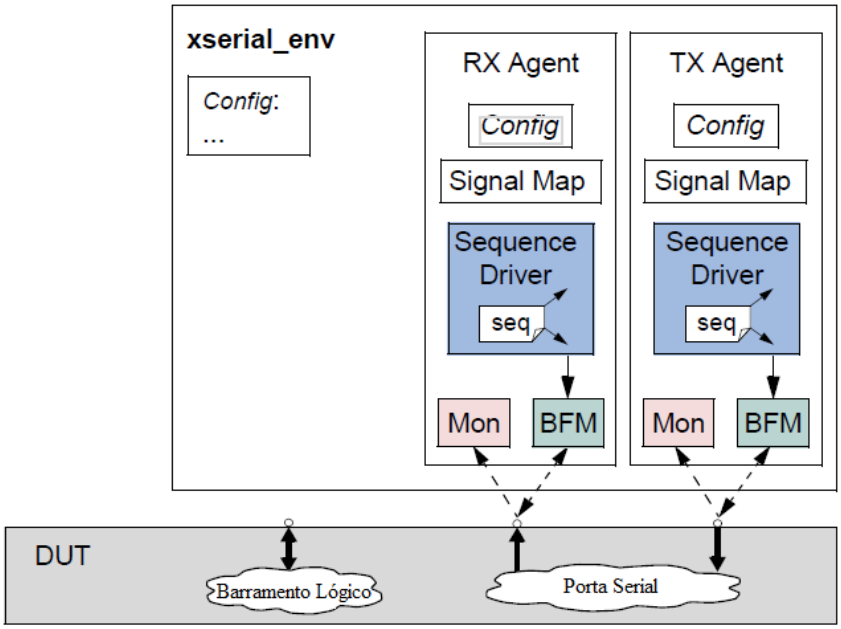

Figura 12: Diagrama de blocos de um *testbench* *e*RM [24]

O DUT da figura 12 possui duas interfaces externas: um barramento lógico e uma interface serial. Como cada uma destas interfaces pode interagir com o meio externo, é

necessário acoplar um *e*VC para exercitar e interagir com cada uma delas.

Olhando inicialmente para a interface serial, observa-se que ela é composta por uma porta que recebe dados e outra que transmite dados. O *e*VC acoplado a esta interface (xserial_env), possui dois Agentes (RX *Agent* e TX *Agent*) instanciados em seu interior e dentro deles existem basicamente cinco subcomponentes:

- ***Config*:** Um conjunto de campos que permitem a configuração do comportamento do agente;
- ***Signals*:** Um conjunto que representa os sinais de *hardware* que o agente precisa acessar para interagir com o DUT;
- ***Sequence Driver*:** Uma unidade utilizada para coordenar os cenário dos testes em execução implementados como sequencias;
- ***BFM (Bus Functional Model)*:** uma unidade que interage com o DUT e tanto coleta quanto envia sinais ao DUT;
- ***Monitor*:** Um unidade passiva que apenas coleta sinais do DUT. Os monitores podem emitir eventos quando captam alguma coisa interessante acontecendo no DUT ou na interface de comunicação. Eles também podem checar o correto funcionamento do DUT e coletar dados de cobertura.

Com estes componentes é possível gerar estímulos, enviá-los para o DUT, coletar dados de funcionamento e checá-los dinamicamente, assim executando um teste de verificação. Notaremos adiante que a estrutura do *testbench* das outras metodologias se assemelha muito à descrita aqui.

4.2 *Verification Manual Methodology* (VMM)

A VMM foi elaborada pela ARM em conjunto com a Synopsys em 2005. Ela já foi implementada baseando-se na então emergente linguagem SystemVerilog e possui uma hierarquia de classes OO distribuída em quatro bibliotecas: VMM *Standard*, VMM *Checker*, XVC *Standard* e *Framework* [25]. A premissa desta metodologia é de que ela poderia, finalmente, permitir ao usuário tirar todo proveito possível das características do SystemVerilog de uma maneira concisa. Então, os engenheiros de verificação poderiam usufruir de asserções, componentes reutilizáveis, *testbenches* dinâmicos, cobertura, análise formal ou outras tecnologias avançadas, todas nativas do SystemVerilog [11].

A VMM também está preocupada com a reutilização. Ela utiliza uma arquitetura em camadas, como mostra a figura 13, a fim de possibilitar a construção de ambientes de verificação comuns que facilitam a reutilização e extensão para tirar vantagem da

automação. Esta abordagem possibilita aproveitar componentes comuns entre projetos [26].

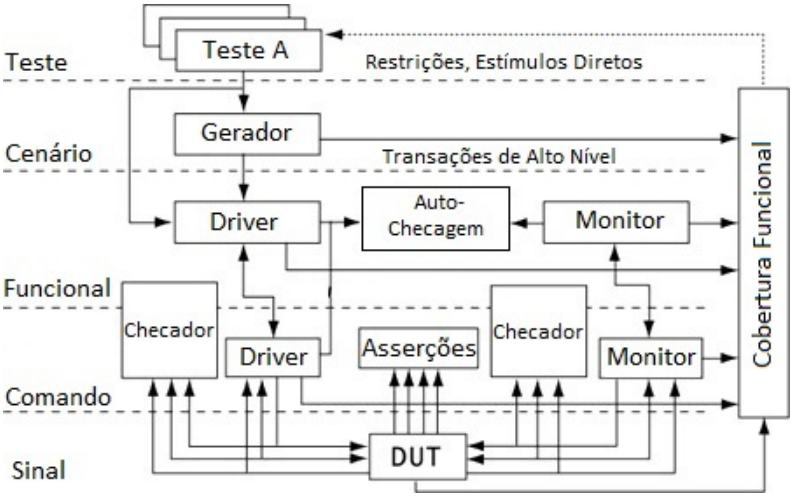

Figura 13: Diagrama de blocos e camadas de um típico *testbench* implementado com a metodologia VMM [26]

O *testbench* da figura 13 possui as seguintes camadas:

- A camada mais baixa é o nível de sinal que conecta o *testbench* com o modelo em RTL. Ela consiste basicamente na interface, no sinal de relógio e das entradas e saídas.

- A camada de comando contém componentes de baixo nível, tais como o *Driver* e o Monitor, bem como as Asserções e Checadores que checam o funcionamento do DUT. Esta camada disponibiliza uma interface a nível de transações com o nível superior e controla os pinos físicos do DUT através da camada de sinais.

- A camada funcional contém *Drivers* e Monitores de alto nível, bem como componentes de auto-checagem que determinam quando o teste foi bem sucedido ou não. Os *Drivers* de alto nível também são conhecidos como *Transactors*, pois passam as transações para o nível imediatamente abaixo.

- A camada de cenário utiliza Geradores para produzir sequências de transações que são aplicadas na camada funcional. O Gerador possui uma série de ajustes ou cenários especificados pelo nível superior. A randomização do teste é introduzida nesta camada.

- Finalmente, a camada de mais alto nível é onde os testes estão alocados. Diferentes testes podem definir novas sequências de transações e até mesmo sincronizar várias sequências. Além disto, é possível gerar estímulos que vão direto para a camada funcional, ignorando a camada de cenário.

Uma grande evolução na VMM é a existência de suporte para o padrão TLM-2.0 [27]. Como descrito anteriormente, há camadas do ambiente de verificação que se comunicam através de transações. Logicamente, isto possibilita a execução de testes em modelos de alto nível, construídos em nível transacional. Desta forma, as duas camadas de mais baixo nível podem ser descartadas e as transações passam diretamente para o DUT, como mostra a figura 14.

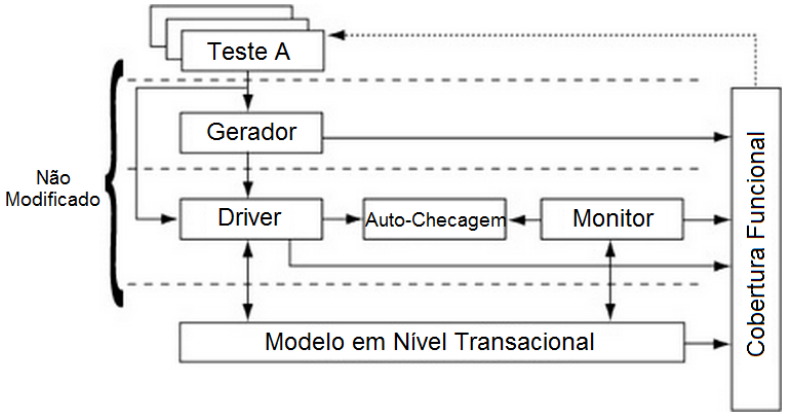

Figura 14: Diagrama de blocos de um *testbench* implementado com a metodologia VMM voltado para modelos TLM [26]

Outra característica importante na VMM é a presença de fases durante a simulação. Estas fases são uma série de métodos virtuais que executam uma operação muito bem definida durante o ciclo de operação do *testbench* [27]. As fases e as operações que elas devem executar são:

1. ***gen_config()***: Cria objetos de dados que servirão para configurar o *testbench*;
2. ***build()***: Cria instâncias de todos os componentes necessários para o *testbench*;
3. ***reset_dut(), cfg_dut()***: Inicializa o DUT;
4. ***start()***: Geradores e *Transactors* iniciam a geração e transmissão de transações, respectivamente;
5. ***wait_for_end()***: Uma fase intermediária que antecede o fim da simulação, permitindo o *testbench* continuar sua execução até que os testes tenham terminado;
6. ***stop(), cleanup()***: Para a execução dos Geradores e deixa o DUT sem atividade. Também salva todos os arquivos para que os resultados possam ser conferidos posteriormente;
7. ***report()***: Encerra a simulação mostrando um sumário contendo dados da simulação.

Posteriormente, o código fonte da VMM foi doado pela Synopsys à Accellera para acelerar o desenvolvimento de interoperabilidade de padrões de verificação que serão discutidos na seção referente à UVM. Seu código fonte é aberto encontra-se disponível sob a licença Apache 2.0.

4.3 *Open Verification Methodology* (OVM)

A metodologia OVM é baseada em uma biblioteca de hierarquia de classes do SystemVerilog e seu código fonte é aberto e disponibilizado sob a licença Apache 2.0. Esta metodologia permite a escrita de ambientes de verificação modulares e altamente reutilizáveis em uma estrutura de montagem *top-down* onde os componentes se comunicam no nível de transação ao longo das camadas.

A OVM foi desenvolvida a partir da soma dos esforços da Mentor Graphics e Cadence e não foi feita a partir do zero. Ela é resultado da união e adaptação de outras duas metodologias já existentes, a URM (*Universal Reuse Methodology*) da Cadence e a AVM (*Advanced Verification Methodology*) da Mentor Graphics, que foi o primeiro trabalho da empresa para adentrar no mercado da verificação funcional.

A biblioteca OVM é baseada em três tipos principais de classes: *ovm_object*, *ovm_component* e *ovm_transaction*. Todas as classes derivam da classe mais básica, a *ovm_object*, e assim constroem seus próprios métodos. A classe *ovm_component* engloba todos componentes utilizados para montar o ambiente de verificação: *Monitor*, *Scoreboard*, *Driver*, *Test*, *Env*, *Agent* e *Sequencer*. Já a classe *ovm_transaction* é responsável pelo transporte de dados entre os componentes. Através dela é possível construir sequências de transações que irão exercitar o DUT [28].

A figura 15 mostra uma estrutura de como um *testbench* pode ser montado com a biblioteca OVM. Aqui, reaparece o conceito de Agente, um componente que contém dentro dele basicamente três outros componentes: Monitor, *Driver* e Sequenciador. Deve haver um Agente conectado a cada interface de comunicação do DUT, sendo que os Sequenciadores geram sequências de transações para que os *Drivers* enviem sinais para o DUT. Concomitantemente, os Monitores coletam dados de resposta do DUT. Podem ainda existir Agentes ditos passivos, pois eles apenas coletam dados de uma interface. Isto significa que em seu interior existe apenas um Monitor [28].

Ainda dentro do *Env*, existem outros dois componentes chamados de *Scoreboard* (Marcador) e *Subscriber* (Assinante). O primeiro recebe e compara dados de ambos os monitores, verificando o funcionamento do DUT; o segundo coleta dados de cobertura e indica quando os testes já podem ser finalizados.

O ambiente de verificação é instanciado dentro de um componente chamado *Test*.

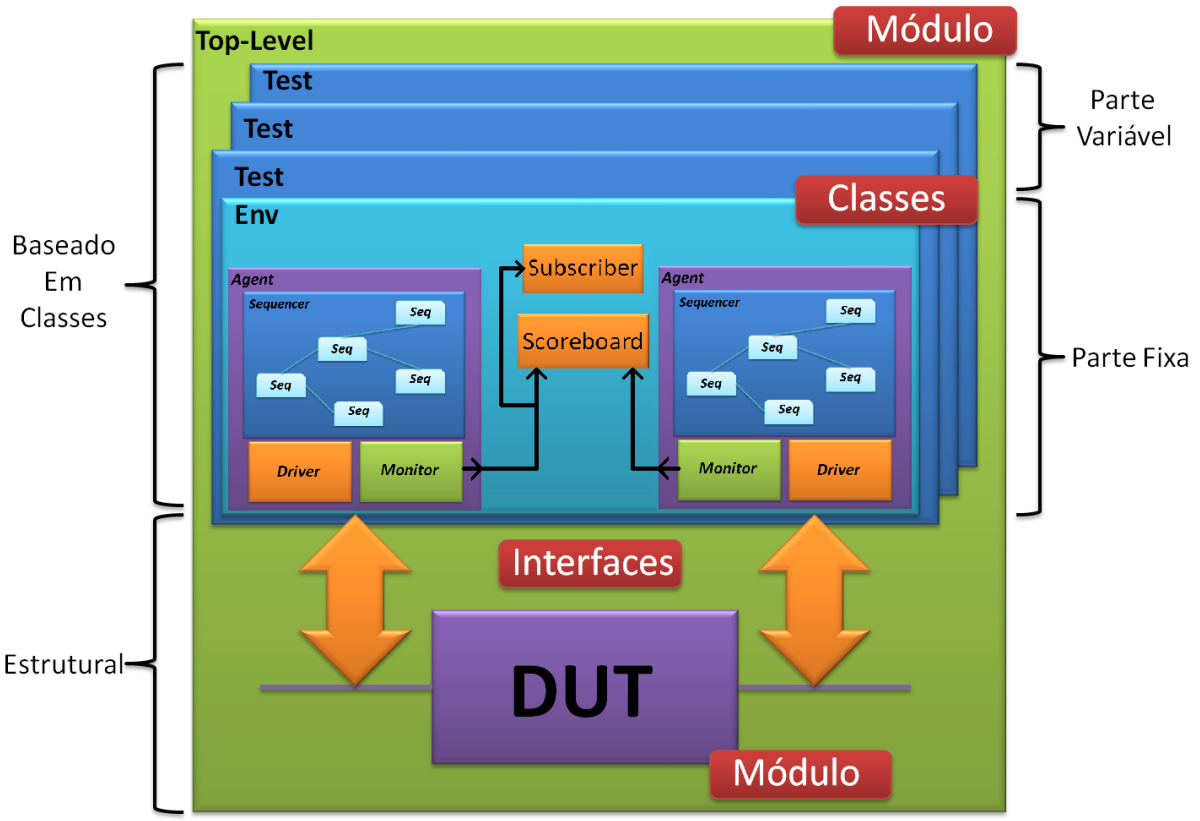

Figura 15: Diagrama de blocos de um *testbench* OVM

O Teste deve ser a parte reconfigurável do *testbench* e é ele que irá dizer quais tipos de cenários devem ser criados para estimular o DUT durante a simulação.

O ambiente de verificação é interligado com o DUT através de uma interface, sendo que ambos são estruturais. E para que classes, módulos e interfaces funcionem adequadamente juntos, deve-se encapsular tudo dentro de um módulo *Top-Level*.

A OVM também implementa o conceito de fases durante o processo de simulação. Segundo o Guia do Usuário da OVM [28], existem 8 fases:

1. ***build()***: Cria os componentes do ambiente de verificação, partindo o nível mais alto para o mais baixo de instanciação;

2. ***connect()***: Conecta os componentes entre si;

3. ***end_of_elaboration()***: Permite fazer ajustes finais no ambiente de verificação;

4. ***start_of_simulation***: Esta fase providencia uma oportunidade de imprimir na tela de simulação informações uteis para o usuário;

5. ***run()***: Este método é o único que consome tempo. Todas as tarefas *run()* dos componentes executam em paralelo;

6. ***extract()***: Esta fase serve para extrair resultados da simulação antes de conferi-los;
7. ***check()***: Fase utilizada para validar os dados obtidos da simulação;
8. ***report()***: Entrega os resultados finais dos testes em arquivos ou na tela do simulador.

É notório a versatilidade deste tipo de abordagem. Pode-se ainda notar que existe a mesma distribuição em camadas bem determinadas como visto na VMM. O fato do código fonte ser aberto e de existir uma forte documentação e vários exemplos disponíveis, tornam a OVM uma alternativa muito boa para empresas que desejam migrar de metodologia. A OVM obteve uma abrangência ainda maior quando a Cadence e a Mentor Graphics disponibilizaram a biblioteca OVM-ML (OVM *Mixed Languages*), que dá suporte a uma gama de HDLs, incluindo o SystemC. Porém, esta biblioteca é de difícil utilização, possui pouco material explicativo e não dá total suporte ao padrão TLM-2.0.

Apesar da OVM ser uma metodologia aberta, ela não é suportada por qualquer compilador SystemVerilog. Inicialmente ela possuía suporte dos compiladores Incisive da Cadence e QuestaSim da Mentor Graphics. Com a grande adoção do mercado, ganhou, posteriormente, suporte de compiladores como o Riviera-PRO da Aldec.

4.4 *Universal Verification Methodology* (UVM)

A UVM baseia-se na OVM e na VMM e constitui uma poderosa ferramenta de verificação desenvolvida pela Cadence, Mentor Graphics e Synopsys sob a luz do grupo ASI (*Accellera System Initiative*) que possui como membros os principais líderes da indústria de EDA, IP e semicondutores do mundo, tais como Intel, AMD, ARM e Texas Instruments. Este padrão, assim como suas antecessoras, possui código aberto, baseado em SystemVerilog e licença Apache 2.0.

A UVM surgiu no ano de 2010 com o intuito unificar as metodologias de verificação na indústria de forma que se criasse um padrão aceitável por todos. Isto recai principalmente sobre a verificação de IPs que necessitam ser de alta confiança. Assim, se um IP é verificado segundo uma metodologia acessível e aceita por todos, consequentemente ele terá maior visibilidade e confiança no mercado.

O pacote UVM acompanha não somente o código fonte, mas também uma extensa documentação contendo o manual do usuário e a referência de classes explicando cuidadosamente cada classe e seus respectivos métodos. Também é possível encontrar dezenas de exemplos abordando os diversos temas inerentes a esta metodologia. Outras páginas da internet, tais como a Verification Academy e Doulos, disponibilizam treinamento gratuito em forma de vídeo-aulas e artigos.

A estrutura do *testbench* continua praticamente idêntica ao padrão OVM bem como a hierarquia de classes. Na nota de lançamento (*release note*) que acompanha o pacote [29], são listadas todas as alterações que foram feitas, dentre elas estão algumas listadas abaixo:

- A compatibilidade com a URM e AVM (bases da OVM) foi retirada;
- O nome das classes passaram de *ovm_<nome_da_classe>* para *uvm_<nome_da_classe>*;
- Os métodos das fases mudaram o nome de *<nome_da_fase>* para *<nome_da_fase>_phase*;
- Doze fases foram inserida durante etapa *run* do processo de simulação para obter maior controle do processo;
- Suporte para TLM-2.0.

Assim como a OVM possui uma biblioteca para interoperabilidade com outras linguagens, a UVM também ganhou uma biblioteca adicional desenvolvida pela Mentor Graphics chamada UVM *Connect* (UVMC) [30]. A UVMC é específica para conectar porções de código em SystemVerilog e SystemC. Assim, pode-se criar um ambiente de verificação utilizando a UVM para validar modelos descritos em SystemC.

Por fim, pode-se dizer que a UVM, mesmo que recente, tem quebrado um paradigma muito grande do mercado EDA. O grupo ASI tem impulsionado fortemente a migração das metodologias antigas para a UVM. Isto traz os benefícios de se ter uma ferramenta poderosa em mãos, com amplo conteúdo didático disponível pra os iniciantes e pleno suporte para as duas metodologias mais utilizadas no mercado anteriormente (OVM e VMM), além de estar enquadrado em um padrão aceito pelos líderes de mercado. Contudo, ainda há o problema de estar preso a ferramentas disponibilizadas pela Cadence, Mentor Graphics e Synopsys. Alguns outros compiladores já estão dando suporte a esta metodologia, mas nada se compara a um código de fato aberto, tal como o SystemC. Criar uma hegemonia no mercado pode não ser interessante, pois esfria a pesquisa e o desenvolvimento de novas soluções que podem inovar e trazer melhores benefícios.

4.5 Adequabilidade das Metodologias a Modelos Transacionais

Para tornar mais interativo o entendimento, é apresentado na tabela 1 um quadro comparativo entre as metodologias descritas anteriormente. Este quadro visa expor as características pertinentes para executar a verificação funcional de um modelo transacional de processador e analisar qual das metodologias melhor atende aos requisitos.

Metodologia	**Linguagem Nativa**	**CDV**	**HDLs Suportadas**	**Compiladores**	**Suporte TLM-2.0**
*e*RM	*e*	Sim	Verilog e VHDL	Specman	Não
VMM	SystemVerilog	Sim	Verilog, VHDL, SystemVerilog, OpenVera e SystemC	VCS e Riviera PRO	Sim
OVM	SystemVerilog	Sim	Verilog, VHDL, SystemVerilog, *e* e SystemC	QuestaSim, Incisive e Riviera PRO	Sim
UVM	SystemVerilog	Sim	Verilog, VHDL, SystemVerilog, *e* e SystemC	QuestaSim, Incisive, VCS e Riviera PRO	Sim

Tabela 1: Quadro comparativo entre as principais metodologias de verificação

A *e*RM é uma metodologia que possui conceitos extremamente importantes para a verificação funcional, mas sua utilização fica limitada somente a modelos descritos em linguagem Verilog e VHDL. Além disso, observa-se que esta metodologia evoluiu naturalmente para a URM. Então, parte-se do pressuposto de que, se algo evolui, evolui para melhor e para se adaptar à realidade e às mudanças e tendências do mercado.

A VMM sem dúvida foi uma das metodologias mais adotadas no mercado durante vários anos. Ela é baseada em SystemVerilog (a linguagem dominante e mais adequada para verificação) e possui conceitos extremamente robustos de reutilização, divisão do *testbench* em camadas e integrabilidade dos componentes de verificação. Contudo, esta metodologia, além de um pouco defasada no tempo, perdeu espaço no mercado para a poderosa OVM, a qual veio a unir forças posteriormente para gerar a UVM.

A OVM talvez seja a principal concorrente da UVM e a que mais gerou dúvidas no processo de seleção da adequabilidade. Esta metodologia consagrou-se como uma poderosa ferramenta de verificação dada a sua consistência e logo se tornou líder devido ao suporte nativo pelas ferramentas da Cadence e Mentor Graphics, que já eram bastante utilizadas no mercado. O código aberto em SystemVerilog e o vasto conteúdo didático disponível geraram uma inevitável tendência de migração das outras metodologias para esta. A grande falha desta metodologia é que ela somente consegue suportar o padrão TLM-2.0 através da biblioteca OVM-ML, que, como dito anteriormente, é de difícil utilização.

A última metodologia, a UVM, atendeu, sem sombra de dúvidas, todos os requisitos desejados para a realização deste trabalho. O simples fato da união das então metodologias mais utilizadas na indústria (a saber, OVM e VMM) já diz muito sobre a força que esta metodologia possui. Além de ser uma metodologia altamente aceita no mercado devido à influência do grupo ASI, possui código aberto, uma infinidade de exemplos e material didático, além de suporte para suas duas metodologias precedentes.

A UVM integra todas as boas práticas da OVM e VMM, tais como a construção de *testbenches* reutilizáveis, reconfiguráveis, multilinguagem e robustos, suporte para CDV, suporte nativo para o padrão TLM-2.0, além de um aperfeiçoamento dos métodos das classes e das fases de simulação já existentes na OVM.

5 Modelagem de Sistemas

Neste capítulo serão introduzidos alguns conceitos complementares para o escopo deste trabalho que auxiliarão na compressão do processo de implementação e verificação de um dispositivo. Primeiramente, serão mostrados os níveis de abstração de um projeto segundo o diagrama de Gajski-Kuhn. Logo em seguida, será dada uma pequena introdução ao conceito da metodologia em nível de sistema eletrônico (ESL) e, por fim, será explicado o conceito da modelagem em nível de transação (TLM).

5.1 Níveis de Abstração na Modelagem de Sistemas Eletrônicos

A carta de Gajski-Kuhn, ou diagrama em Y, descreve as diferentes perspectivas em um modelo de *hardware*. Este diagrama permite visualizar tanto as perspectivas do modelo como também suas hierarquias. Ele é largamente utilizado em projetos de VHDL e também pode dar uma noção dos níveis de abstração que serão de extrema importância para a próxima seção [31].

Esta carta mostra os diferentes estágios do desenvolvimento de um circuito integrado. De acordo com este modelo, o desenvolvimento de um *hardware* é observado em meio a três domínios e cinco níveis hierárquicos. Os três domínios são representados pelos eixos radiais produzindo o formato de Y e os cinco níveis hierárquicos são representados pelos círculos concêntricos. O círculo mais externo possui maior abstração e o círculo mais interno possui maior refinamento do projeto [31]. O diagrama pode ser visualizado na figura 16.

Os domínios deste diagrama são explicados a seguir:

1. **Domínio Comportamental**: descreve o comportamento funcional e temporal de um sistema.

2. **Domínio Estrutural**: um sistema é montado a partir de um conjunto de subsistemas. Neste domínio, os diferentes subsistemas e suas interconexões são listadas para cada nível de abstração.

3. **Domínio Geométrico ou Físico**: aqui, as propriedades geométricas do sistema e de seus subsistemas são consideradas. Então, há informação detalhadas sobre o tamanho, forma e posicionamento físico de cada componente.

E os níveis de abstração ou de refinamento são detalhados a seguir:

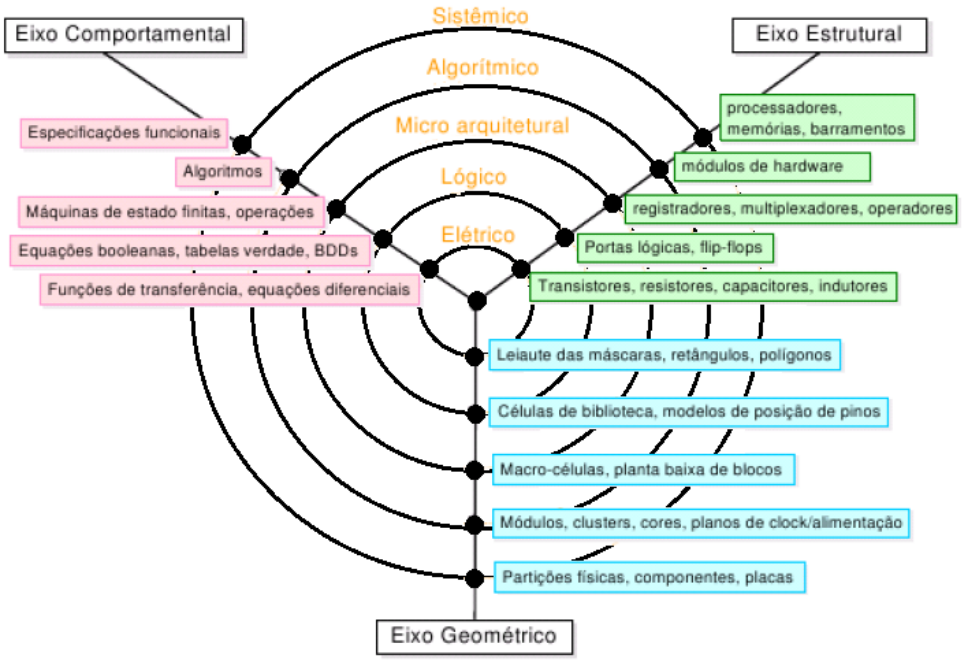

Figura 16: Carta de Gajski-Kuhn ou Diagrama em Y [31]

1. **Nível de sistema**: no nível de sistema, propriedades básicas de um sistema eletrônico são definidas. Para a descrição comportamental, diagramas de blocos são utilizados fazendo abstração dos sinais e sua resposta no tempo. Exemplos de blocos utilizados no domínio estrutural são CPU, *chips* de memória, etc.

2. **Nível de Algoritmo**: o nível de algoritmo é descrito pela definição dos algoritmos utilizados simultaneamente (sinais, *l*oops, variáveis, decisões). No domínio estrutural, blocos como ULA são utilizados.

3. **Nível RTL**: o nível RTL possui uma abstração menor e é mais detalhado, onde o comportamento entre os registradores em uso e as unidades lógicas é descrito. Aqui, estruturas de dados e o fluxo de dados são definidos. Na visão geométrica, a etapa de posicionamento e roteamento dos componentes é descrita.

4. **Nível Lógico**: o nível lógico é descrito no domínio comportamental através de equações Booleanas. Na visão estrutural, isto é representado com portas lógicas e flip-flops. No domínio geométrico, o nível lógico é representado por células padrão.

5. **Nível de Circuito**: O comportamento do nível de circuito é descrito por equações matemáticas usando equações diferenciais ou equações lógicas. Isto corresponde a transistores e capacitores ou até mesmo redes cristalinas.

A figura 17 mostra de forma simples e objetiva como cada nível de abstração pode representar um sistema de acordo com o que foi descrito acima. No nível mais alto, o nível de sistema, o projeto é representado apenas com caixas pretas, mostrando

apenas a interconexão entre estes blocos maiores. O nível de algoritmo define a lógica comportamental do sistema e é descrito através de código. O nível RTL descreve blocos de circuitos mais específicos, tais como multiplexadores, ULAs, memórias e registradores e ainda mostra o caminho para onde a informação está sendo transferida através de setas. O próximo nível, o nível lógico, representa o sistema utilizando portas lógicas e flip-flops e sua interconexão detalhada. Por fim, o nível mais baixo e mais detalhado mostra todos os transistores e componentes discretos que incorporam o sistema eletrônicos. No domínio geométrico, este nível é representado pelo *layout* do circuito através das formas geométricas que serão construídas em silício.

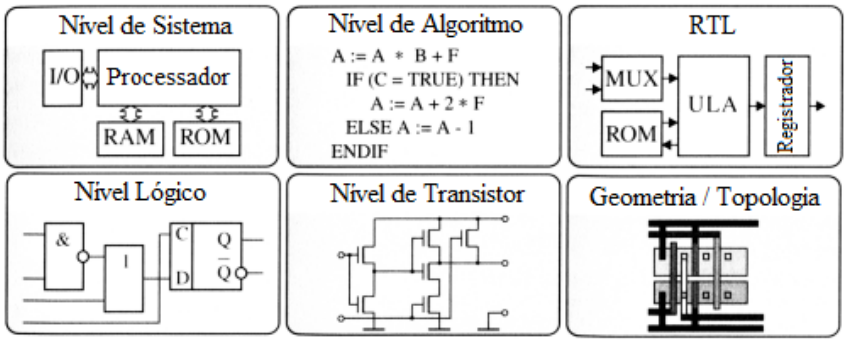

Figura 17: Representação de um sistema visto dos diferentes níveis de abstração [32]

A questão no desenvolvimento de *hardware* é frequentemente um problema do tipo *top-down*. Isto pode ser observado a partir dos três domínios - comportamental, estrutural e físico - que vão em direção ao centro no diagrama, reduzindo o nível de abstração ao se acrescentar mais detalhes. O projetista pode escolher uma das perspectivas e depois mudar de uma visão para outra. Vale ressaltar que o processo de modelagem não deve necessariamente seguir uma sequência específica no diagrama Y.

A próxima seção irá utilizar os conceitos aqui expostos para ilustrar a atual metodologia de projeto e verificação utilizada pela indústria, tendo em vista a atual realidade dos sistemas complexos e de alto desempenho.

5.2 ESL - *Electronic System-Level*

De uma forma geral, ESL é uma abordagem atualmente adotada nas maiores companhias de desenvolvimento de SoC e está sendo cada vez mais utilizada para a modelagem de sistemas. Trata-se de uma metodologia que se desdobra em um conjunto de metodologias complementares que permitem a modelagem de sistemas embarcados, verificação e depuração através da implementação do *hardware* e *software* de SoCs, *system-on-FPGA* (*Fiel-Programmable Gate Array*) e sistemas multiplacas completos. ESL pode ser realizada através do uso do SystemC como linguagem de modelagem [33].

A nova realidade trazida pelos SoCs decorrente da evolução da tecnologia VLSI (*Very-Large-Scale Integration*), juntamente com o crescimento contínuo da complexidade dos circuitos, tem prejudicado os prazos de mercado. A procura por alta produtividade envolve uma combinação das seguintes noções [33]:

- **Abstração**: na etapa de implementação, o nível de representação é elevado para lidar com a grande complexidade do modelo. Como consequência, o fluxo de projeto passa por várias camadas e estilos de representação. As representação de *hardware* têm se elevado através dos níveis físico, de circuito, de portas lógicas, RTL e funcional/comportamental.
- **Reutilização**: a reutilização já foi um assunto bastante discutido na primeira parte deste trabalho. Componentes previamente desenvolvidos são reutilizados em um novo projeto a fim de ganhar tempo e economizar gastos.
- **Automação**: para superar a tendência a erros e a demorada natureza do trabalho manual, os engenheiros de implementação dependem das ferramentas EDA. Para refinar a representação do modelo dos níveis mais altos de abstração para os mais baixos, ferramentas de síntese podem ser utilizadas. Para conferir a equivalência de funcionalidade entre os sucessivos níveis de representação, ferramentas de verificação automática estão disponíveis. A este procedimento, dá-se o nome de *High-Level Synthesis* (HLS).
- **Exploração**: a análise de soluções alternativas para um determinado projeto em relação à área, performance e potência em um dado nível de abstração reduz a probabilidade do modelo não atender os requisitos após o refinamento através dos níveis de representação, levando a uma nova rodada para correções.

Estes aspectos conduzem ao conceito de Nível de Sistema Eletrônico, um termo genérico para um conjunto de abstrações, os quais são adequados para a representação de projeto de SoCs [34].

5.2.1 Modelo de Taxonomia

Quando novas tecnologias surgem no mercado, há uma divergência entre a terminologia utilizada por diferentes grupos. Para minimizar este problema que acontece com frequência tanto no meio acadêmico quanto na indústria, Martin, Bailey e Piziali [33] propôs em sua obra um modelo de taxonomia para definir termos e classificações bastante utilizadas. Portanto, taxonomia é a caracterização de objetos ou conceitos baseada na relação existente entre eles. A taxonomia pode ser representada em um gráfico hierárquico ou em uma tabela de atributos, onde cada atributo identifica um elemento particular da diferenciação.

O modelo taxonômico do ESL é composto por quatro eixos principais: temporal, dados, funcional e estrutural. Estes quatro eixos não são completamente ortogonais e o eixo da funcionalidade reflete sobre alguns conceitos do eixo temporal e eixo de dados [33]. Cada um destes eixos pode ser observado na figura 18. Quanto mais à direita de um eixo encontra-se um ponto, menor será o seu detalhamento e, consequentemente, maior será o seu nível de abstração. Já os pontos situados mais à esquerda dos eixos possuem menor abstração e maior detalhamento.

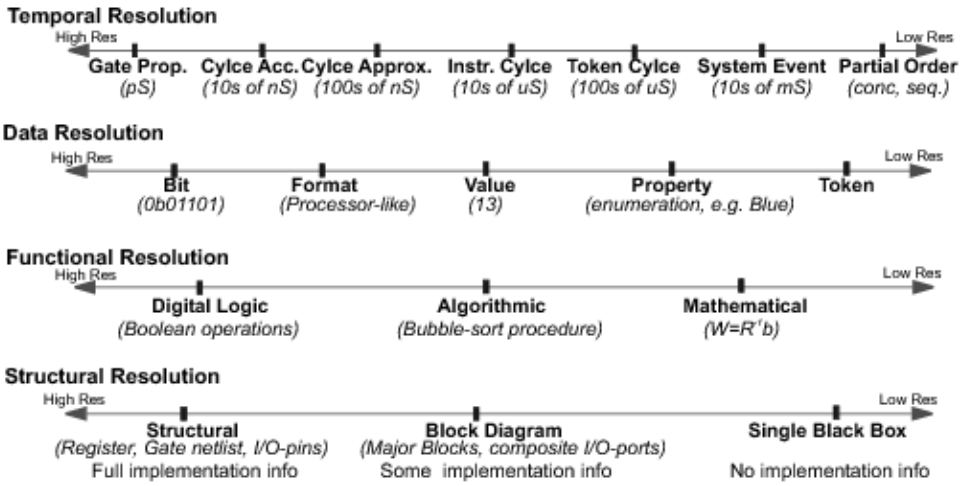

Figura 18: Representação dos quatro eixos de abstração ESL

EIXO TEMPORAL

Este eixo define o tempo em um modelo. Os pontos definidos sobre este eixo são "eventos parcialmente ordenados", "evento de sistema", "ciclo de *token*", "ciclo de instrução", "aproximação de ciclo", "precisão de ciclo" e "precisão de propagação de porta".

O ponto mais abstrato definido na escala é "evento parcialmente ordenado". Neste ponto, sabe-se quando algum evento irá começar e terminar apenas pelo seu relacionamento com o início e o término de outro evento em uma execução particular. As interfaces bloqueantes do TLM se encaixam aqui. Em contrapartida, o ponto mais preciso e menos abstrato na escala do eixo é "precisão de propagação de porta". Neste ponto, o período de *clock* é conhecido e deve-se levar em consideração até os atrasos de propagação nas portas lógicas. Modelos descritos com esta abordagem são extremamente precisos.

EIXO DE DADOS

O eixo dos dados define o nível de precisão da informação. Poucos pontos são definidos neste eixo: "*token*", "propriedade", "valor", "formato" e "bits". O ponto mais abstrato, o *token*, indica que algum dado está se movendo pelo sistema, mas o seu tamanho e o seu conteúdo são desconhecidos. O ponto menos abstrato, "bits", fornece o mapeamento de valores do *hardware* que irá armazená-lo.

EIXO FUNCIONAL

Este eixo indica a precisão das operações. Apenas três pontos são definidos: “relação matemática”, “algoritmo” e “lógica digital”. A relação matemática define precedência, mas não define ordem. Um algoritmo pode ser selecionado para implementar a relação que define em qual ordem as coisas serão feitas. Eventualmente, a implementação de uma lógica digital/Booleana pode instanciar as unidades funcionais que irão ser usadas no algoritmo.

Este eixo não é completamente independente dos demais porque ele carrega noções de tempo, dados e também de estrutura.

EIXO ESTRUTURAL

Neste eixo, a quantidade de detalhes estruturais presentes no modelo é marcada, indicando portanto o quão próximo o modelo está da implementação. Esta escala não fornece um meio de verificar se a estrutura do modelo é diferente da estrutura da implementação final. O modelo de taxonomia sugere que, se um bloco utilizado não obedece à estrutura da implementação final, ele é considerado uma *black-box* sem estrutura interna.

5.3 TLM - *Transaction Level Modeling*

O padrão TLM é uma das tecnologias básicas que torna o ESL prático. Esta metodologia foi criada para permitir a representação de um modelo em um nível intermediário de abstração entre o RTL e as especificações de requisitos do modelo. Para entender o TLM, é preciso, primeiramente, entender a terminologia que descreve os níveis de abstração, bem como os modos em que os modelos serão utilizados [35].

O TLM se refere a um conjunto de elementos de código utilizados para criar modelos no nível de transações. TLM-1.0 e TLM-2.0 são dois padrões os quais foram desenvolvidos pela indústria para construir modelos transacionais. Ambos foram baseados em SystemC e padronizados pelo grupo OSCI (*Open SystemC Initiative*), sendo que o TLM-1.0 foi padronizado em 2005 e o TLM-2.0, em 2009 [36].

O primeiro conceito chave por trás do TLM é que detalhes desnecessários podem ser ignorados nas primeiras fases do fluxo de projeto. Utilizando um alto nível de abstração, o TLM proporciona um grande ganho na velocidade de simulação e na produtividade de modelagem. A principal meta do TLM é se tornar um modelo de referência para equipes trabalhando com *software*, *hardware*, análise de arquitetura e verificação.

O segundo conceito chave é a separação entre computação e comunicação. Em uma representação TLM, os módulos possuem processos concorrentes que executam seus próprios comportamentos, enquanto que a comunicação é realizada através da troca de pacotes entre estes módulos, as quais são chamadas de transações. A comunicação é implementada dentro de canais, omitindo o protocolo dos módulos, mas expondo suas

interfaces.

Embora o TLM seja uma linguagem independente, o SystemC é uma das linguagens que se adequa perfeitamente ao seu estilo de representação, permitindo níveis adequados de abstração e fornecendo elementos para suportar a separação entre computação e comunicação [34].

5.3.1 Níveis de Abstração do TLM

Os níveis de abstração descritos na subseção 5.2.1 podem ser adaptados para um modelo adequado ao TLM, como mostra a figura 19. Observa-se que o padrão TLM se encaixa em diversas posições deste diagrama, mostrando assim, sua abrangência e versatilidade. Na figura[1], o eixo vertical representa o nível de abstração da funcionalidade, enquanto o eixo horizontal representa o nível de abstração das interfaces ou da comunicação. É importante notar que a funcionalidade pode ser refinada independentemente da interface e que a recíproca também é verdadeira. Para cada par de abstração entre comunicação e funcionalidade, existe um caso de uso específico onde o TLM se encaixa [35].

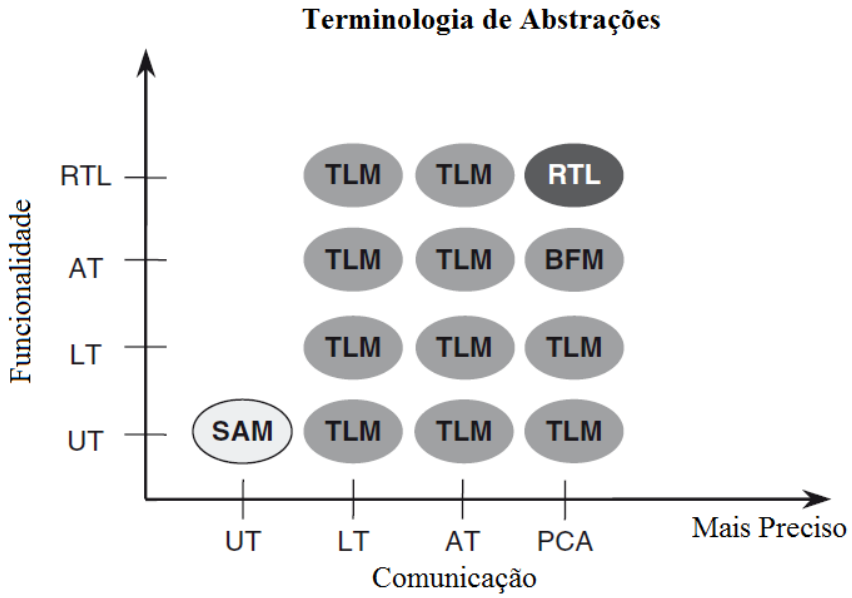

Figura 19: Mapa do modelo TLM em seus níveis de abstração [35]

Segundo o manual do TLM [36], simulações do tipo LT (*loosely-timed*) são adequadas para desenvolvimento de *software*, análise de desempenho de *software* e análise arquitetural. As simulações do tipo AT (*approximately-timed*) também são indicadas para análise arquitetural e para verificação de *hardware*. Nas simulações LT é utilizado o recurso de desacoplamento temporal (*temporal decoupling*), o qual permite que partes do modelo sejam executadas antes para depois sincronizarem o restante do sistema, permi-

[1] O significado de todas as abreviaturas pode ser consultado na lista contida no início do documento.

tindo simulações mais rápidas. No caso das simulações AT, é feita uma subdivisão do tempo de vida de uma transação em diferentes fases [37].

5.3.2 Metodologia TLM

A metodologia TLM pode ser descrita conforme a figura 20 [35]. Este fluxo começa pelo método tradicional utilizado para capturar os requisitos desejados pelo cliente, na imagem descrito como *Requirements Document*. A partir deste documento, um modelo TLM em alto nível é desenvolvido. O trabalho no desenvolvimento de um modelo TLM pode gerar alterações e refinamentos no documento de especificações. A equipe de arquitetura deve capturar as especificações do produto e os parâmetros críticos para a construção deste modelo.

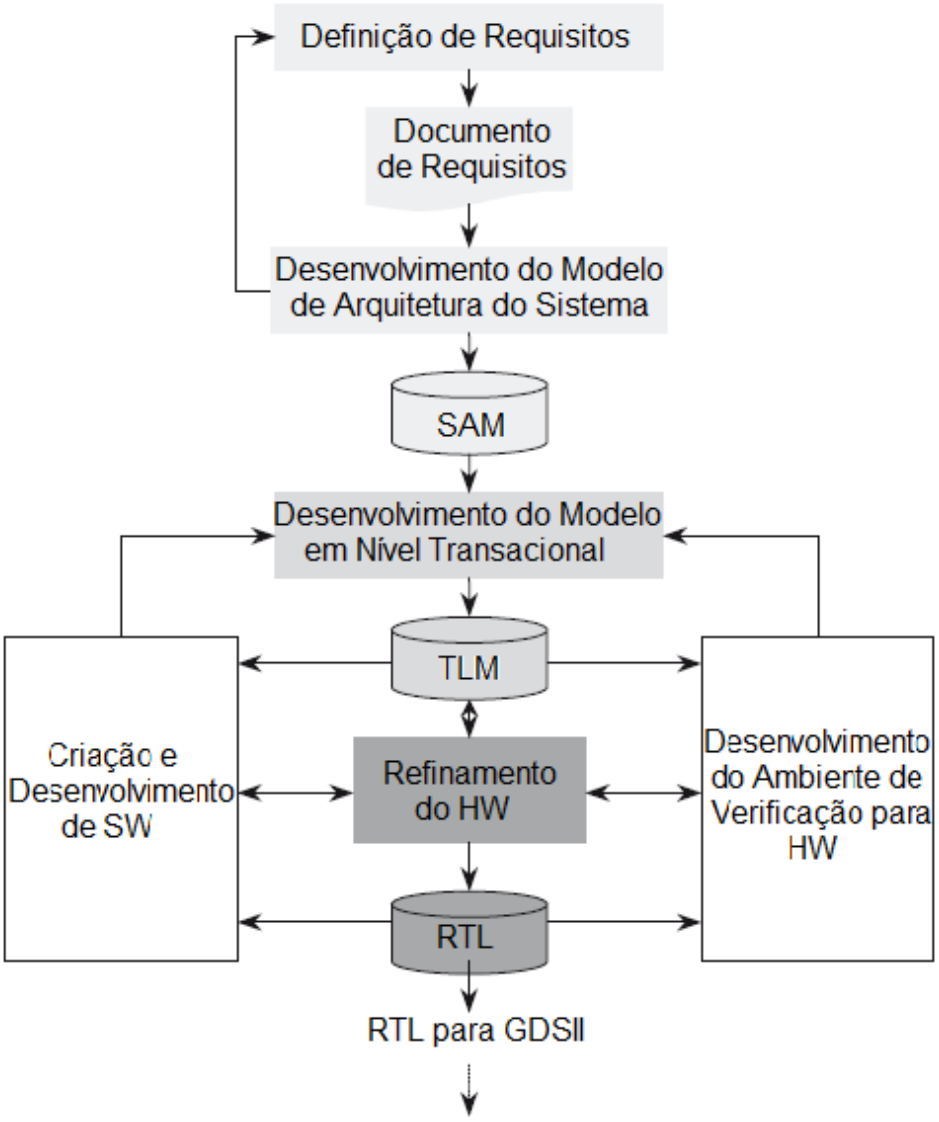

Figura 20: Fluxo da metodologia TLM [35]

Posteriormente, o modelo TLM é refinado em duas direções distintas: 1) desen-

volvimento e modelagem de *software* e 2) desenvolvimento do ambiente de verificação de *hardware*.

O TLM pode ser reutilizado e refinado se as técnicas forem utilizadas corretamente ao longo do fluxo. A seguir, são enumerados os casos de uso do TLM:

1. Modelagem de arquitetura
2. Modelagem de algoritmo
3. Desenvolvimento de plataforma virtual
4. Verificação funcional
5. Refinamento de *hardware*

Aplicando o TLM ao desenvolvimento de um projeto, pode-se obter valiosos benefícios, tais como:

- Desenvolvimento antecipado de *software*, que normalmente é a atividade que limita o desenvolvimento de um sistema
- Elaboração antecipada e melhorada de *testbenches* para verificação funcional de *hardware*
- Um caminho limpo e coerente entre as especificações do cliente e as especificações de *hardware* e *software*

5.3.3 Principais Características do TLM-2.0

Os padrões TLM-1.0 e TLM-2.0 compartilham uma herança comum e a maioria das pessoas que desenvolveram o TLM-1.0 também trabalharam na elaboração do TLM-2.0. Todavia, eles são padrões bastante diferentes. TLM-1.0 é um sistema de transmissão de mensagem. As interfaces podem ser atemporais (UT) ou podem depender do dispositivo alvo para temporização. Nenhuma das interfaces fornece uma anotação explícita do tempo. TLM-2.0 possibilita a transferência de dados e a sincronização entre processos independentes, e é principalmente designado para uma modelagem de alta performance de sistemas baseados em barramentos com mapeamento de memória [38].

A API (do inglês *Application Programming Interface*) do TLM é construída da seguinte forma [38]:

- Uma API genérica que é baseada em *templates* definidas pelo usuário

- Recomendações do tipo de dados de baixo nível a serem usados nas *templates* de usuário

- Estruturas de dados para a API que a tornam totalmente especializada e modelam um barramento genérico com mapeamento de memória

Os modelos descritos em TLM podem ser separados em dois estilos de código, dependendo da temporização que eles devem obedecer: modelo vagamente temporizado (LT) e modelo aproximadamente temporizado (AT) [38]. O modelo vagamente temporizado possui uma baixa dependência entre temporização e dados e são capazes de fornecer informações temporais e o dado requisitado no momento em que a transação é iniciada. Estes modelos não dependem do progresso do tempo para produzir uma resposta, são executados o mais rápido possível e possibilitam inicializar um sistema operacional e executar testes em questão de segundos. Já o modelo aproximadamente temporizado possui uma dependência muito maior entre tempo e dados. Este modelo quebra a transação em um número de fases muito mais aproximado de um protocolo de *hardware*. Em alguns casos, é desejável criar modelos que operem entre os dois estilos, decidindo qual usar através de uma decisão no tempo de execução.

5.4 Estudo de Caso

Tendo visto alguns dos pontos relevantes que ensejam a modelagem de sistemas em um nível de abstração maior, bem como as duas metodologias que lideram este segmento de mercado, deseja-se, agora, descrever o estudo de caso deste trabalho. Para entender a estrutura e arquitetura do processador MIPS Plasma 32 bits, é importante, primeiro, entender como o TLM se materializa através do SystemC. Por isso, será dada uma breve introdução aos conceitos da linguagem antes da implementação ser, de fato, descrita.

5.4.1 SystemC

Estritamente falando, SystemC [12] não é uma linguagem. SystemC é uma biblioteca de classes pertencente a uma linguagem muito bem definida, o C++ [35]. Assim, o uso desta biblioteca facilita o processo de modelagem de *hardware* por possuir tipos de dados próprios para a definição de *hardware* e estruturas e processos que flexibilizam a descrição do paralelismo natural em *hardware*. A base fornece um núcleo de simulação orientado a objeto e qualquer programa pode ser compilado utilizando um compilador C++ e produzir um arquivo executável [37]. E, embora tenha sido desenvolvido para ser uma linguagem a nível de sistemas, pode ser utilizada também para descrever modelos em nível RTL ou em níveis intermediários a estes [39].

Em se tratando das estruturas disponíveis na biblioteca SystemC, o módulo é a principal. Um módulo consiste em um bloco básico usado para particionar um projeto complexo de maneira que ele se torne mais gerenciável. Um módulo pode conter portas para comunicação, processos para descrever seu funcionamento, dados internos, canais de comunicação e outros módulos instanciados em níveis hierárquicos mais baixos. A instanciação de módulos pode ser realizada utilizando regras de instanciação de objetos e algumas adicionais do C++ [37].

A criação de um novo módulo que representa um componente particular do sistema requer herança da classe básica *sc_module*. As macros *SC_CTOR* ou *SC_HAS_PROCESS* declaram o construtor do módulo que registra os processos e a sensitividade a eventos, dentre outras funções [37]. Feito isto, pode-se declarar os métodos e variáveis que irão moldar as funcionalidades requeridas e os mecanismos de comunicação [40].

Os processos implementam o modelo de funcionalidade de um bloco, só podem ser definidos dentro de um módulo e são controlados pelo escalonador da simulação. Existem dois tipos mais usados de processos [40]:

- *SC_METHOD*: são processos declarados como funções do C++ e são chamados em resposta a um determinado evento.
- *SC_THREAD*: também são processos declarados como funções do C++ e são chamados apenas uma vez em uma simulação (normalmente na inicialização) e só podem ser bloqueados pelo comando *wait()*. A execução retorna quando a condição associada ao método *wait()* é satisfeita.

As portas são utilizadas para conectar módulos com a vizinhança, possibilitando o transporte de dados de um ponto a outro. Uma porta é ligada a um canal através de uma interface. As interfaces são classes abstratas que declaram métodos puramente virtuais. Os canais herdam uma ou mais interfaces e são obrigados a prover definições que irão sobrescrever os métodos puramente virtuais. As portas, neste caso, funcionam como um "ponteiro inteligente" que permitem processos chamarem métodos definidos em um canal. SystemC define um conjunto padrão de canais e portas que implementam tanto a comunicação em baixo nível (sinais) quanto em alto nível (FIFO) [40].

Através destas e de outras funcionalidades mais avançadas, a biblioteca SystemC consegue dar suporte ao padrão TLM. A figura 21 mostra o mecanismo básico de conexão e comunicação entre módulos. Um *initiator* é um módulo que inicia uma nova comunicação, um *target* é um módulo que serve como destino final para uma transação e um componente de interconexão é um módulo que repassa uma transação sem modificá-la. As transações são estruturas de dados contendo diversos campos e tipos de dados e são passadas entre *initiators* e *targets* através de chamadas de funções. Para transmitir uma transação são

utilizados *sockets*, representados na figura pelos quadrados com um triângulo dentro. As setas que interligam os *sockets* ditam a direção do fluxo de dados. O *initiator* passa a transação pelo *forward-path* e o *target* responde através do *backward-path* [36].

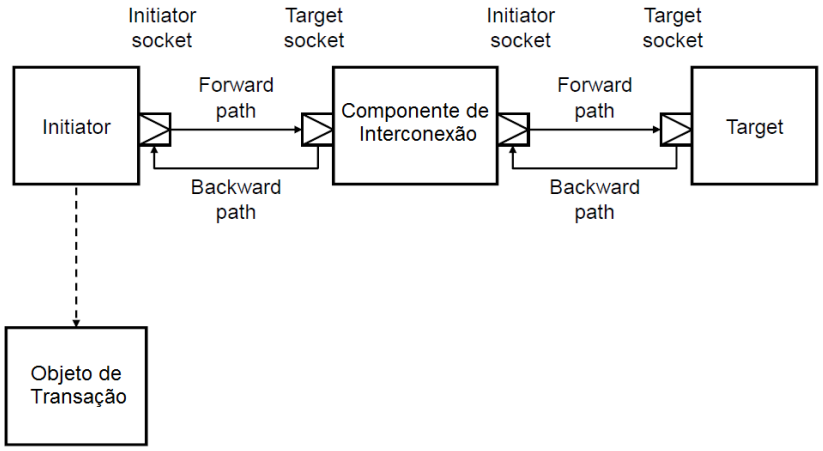

Figura 21: Mecanismos de comunicação do TLM [36]

As interfaces de transporte são o principal tipo de interface utilizada no transporte de transações entre *initiators*, *targets* e componentes de interconexão. Estas interfaces podem ser do tipo bloqueante e não-bloqueante. As primeiras são mais utilizadas no estilo de código LT e são apropriadas quando o *initiator* deseja completar uma transação com o *target* em apenas uma chamada de função. Isto quer dizer que a interface somente modela o início e o fim da transação. O segundo tipo de interface é mais utilizado com o estilo de código AT e são apropriadas quando se deseja modelar detalhadamente a sequência de interações entre *initiator* e *target* durante o curso de cada transação [36].

Por fim, deve-se falar da classe que implementa a estrutura de dados de uma transação no TLM-2.0: a *generic payload*. Ela tem o objetivo de melhorar a interoperabilidade entre modelos de barramento mapeados em memória em dois níveis. Primeiro, propõe um tipo de transação genérica que garante interoperabilidade imediata entre modelos onde os detalhes do barramento não são importantes. Segundo, pode ser utilizada como base para criar modelos detalhados de protocolos específicos de barramento, com a vantagem de reduzir os custos de implementação e aumentando o a velocidade de simulação [36].

A *generic payload* possui atributos padronizados, dentre os quais pode-se citar [37]:

- *Command*: pode ser *read* ou *write* e indica a operação que será feita no barramento;
- *Address*: define o menor valor de endereço no qual os dados são lidos ou escritos;
- *Data*: é um ponteiro que aponta para a estrutura de dados (*buffer*) do *initiator*. Há outro atributo chamado *length* que indica a quantidade de dados a serem copiados ou escritos;

- *Response Status*: indica o *status* da transação.

5.4.2 MIPS Plasma 32 Bits

O processador utilizado aqui como estudo de caso é fruto do trabalho que está sendo desenvolvido por Tiago Trindade da Silva em seu programa de doutorado em Engenharia de Sistemas Eletrônicos e Automação do Departamento de Engenharia Elétrica, Universidade de Brasília. Durante o caminho percorrido até o presente momento, diversas versões deste processador foram desenvolvidas tomando como base o ISA do processador MIPS32 [41], utilizando SystemC, tanto no padrão TLM-1.0 quanto no padrão TLM-2.0. Especificamente, será utilizado o modelo descrito no padrão TLM-2.0.

Este processador foi desenvolvido durante o projeto SoC-SBTVD: Sistema em *Chip* para o Terminal de Acesso do SBTVD (Sistema Brasileiro de Televisão Digital), com o intuito de compor um módulo *parser* para a decodificação de áudio ACC (*Advanced Audio Coding*) em ambiente virtual. Um dos objetivos do trabalho é a definição de um modelo de referência para verificação (*golden model*) que sirva de suporte para modelos RTL, em VHDL, que venham a ser desenvolvidos para a síntese em FPGA e silício.

Como mostra a figura 22, o processador foi construído com uma *pipeline* de cinco estágios, onde cada estágio foi implementado por um módulo SystemC que serão detalhados adiante. O modelo computacional utilizado é o *dataflow*, isto quer dizer que a interface de comunicação entre os módulos implementa somente métodos de escrita, *i. e.* um módulo somente pode escrever dados no módulo subsequente. A única exceção encontra-se nos módulos com acesso à memória, *Fetch* e *Memory Access*, os quais requerem métodos de leitura para acessar dados da memória. Neste caso, foi adotada a arquitetura *Harvard* para acesso simultâneo de dados e instruções na memória de programa.

Cada módulo possui um único processo do tipo *SC_THREAD* em execução que aguarda a ocorrência de uma escrita em sua interface de entrada para entrar em operação. Ao final da execução, a *thread* executa um método de escrita em seu *initiator socket* conectado ao *target socket* do próximo módulo. A interface de comunicação entre estes módulos é do tipo bloqueante, isto quer dizer que o *initiator* aguarda a resposta positiva do *target* para continuar suas tarefas. Os dados transmitidos entre os módulos foram encapsulados em forma de *generic playload* e, apesar do atributo *delay* obrigatório, o valor propagado é “0” (zero), tornando, assim, o modelo atemporal (UT).

Os cinco estágios da *pipeline*, como descritos em Hennessy, Patterson e Asanović [42], são: *Fetch*, *Decode*, *Execute*, *Memory Access* e *Write Back*. A *pipeline* consiste em um mecanismo que explora o paralelismo em meio à sequência de instruções de um processador. Ela permite a sobreposição de múltiplas instruções sendo executadas ao mesmo tempo, em que cada estágio estará trabalhando em uma instrução diferente. Ainda, *pipe-*

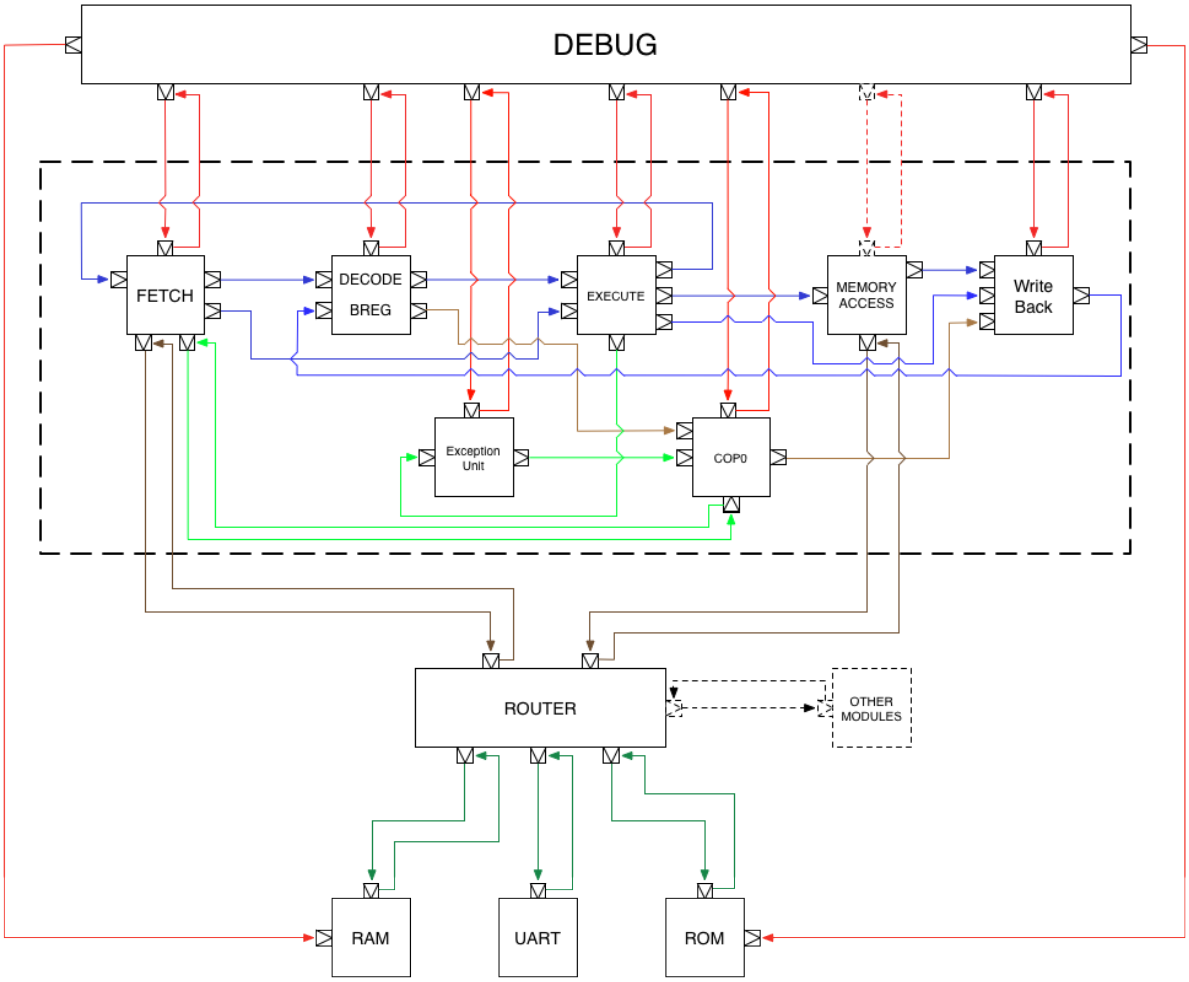

Figura 22: Modelo TLM-2.0 completo do processador MIPS32. Fonte: cortesia de Tiago Trindade da Silva

lines podem conter mais ou menos estágios dependendo da arquitetura e isto pode trazer vantagens e desvantagens para o projeto.

O estágio *Fetch* é responsável por buscar na memória de programa (memória ROM) a próxima instrução a ser executada. Para isto, ele utiliza o *socket* de acesso à memória que passa pelo *Router*. Após buscar a instrução, ela é transmitida para o módulo *Decode*, o qual é responsável por decodificá-la gerando várias informações. Como a arquitetura é de 32 bits, a palavra de memória também é de 32 bits. A figura 23 mostra os três tipos de instruções existentes na arquitetura MIPS, sendo eles: *Immediate*, *Jump* e *Register*.

Instruções do tipo *Immediate* realizam operações com o valor armazenado em um registrador e um valor imediato, informado no código; instruções do tipo *Jump* determinam saltos no programa e, por sua vez, instruções do tipo *Register* realizam operações entre valores armazenados entre dois registradores distintos. A tabela 2 descreve o que cada campo da instrução representa [41].

O estágio *Decode* necessita gerar todos os campos de instrução mesmo que desne-

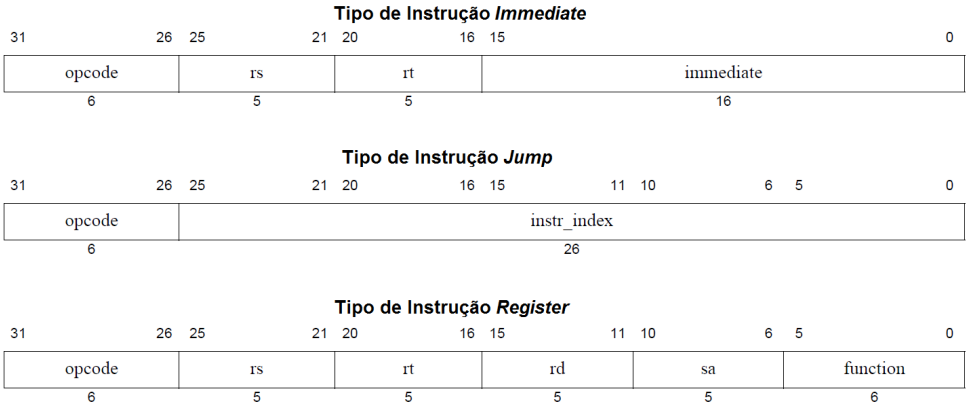

Figura 23: Formatos de instruções do MIPS32 [41]

Campo	Descrição
opcode	Indica o código da operação
rd	Especifica o registrador de destino
rs	Especifica o registrador fonte
rt	Especifica o registrador alvo
immediate	Especifica o valor do imediato utilizado em operações lógico-aritméticas
instr_index	Contém o valor do salto na memória
sa / re	Deslocamento de bits *(shift)*
function	Especifica funções especiais utilizadas com *opcode* primário

Tabela 2: Descrição dos campos da instrução MIPS32 [41]

cessariamente, pois ele não faz distinção do tipo de instrução. Todos estes dados são então transmitidos para o estágio *Execute*, responsável por executar a operação e determinar o endereço da próxima instrução. É neste módulo que ocorre o controle do PC (*program counter*), o qual indica ao módulo *Fetch* o endereço da próxima instrução a ser buscada na memória.

As operações de leitura e escrita na memória de dados (memória RAM) são feitas no módulo *Memory Access*, o qual recebe o endereço, o dado (no caso de escrita) e o tipo da operação (escrita ou leitura). Caso o pacote recebido por este módulo não contenha dados de leitura ou escrita, o mesmo é encaminhado sem nenhuma alteração para o último estágio da *pipeline*, o *Write Back*. Se o pacote recebido pelo módulo *Memory Access* conter dados de escrita ou leitura na memória, a *thread* deste módulo realiza a operação e transfere o resultado para o módulo *Write Back* para que este atualize o banco de registradores.

Vale lembrar que o MIPS32 possui um banco de 32 registradores (GPRs) que está contido no módulo *Decode*. O registrador R0, em particular, pode ser escrito, mas sempre inicia um novo ciclo de instrução com o valor zero. Por este motivo, ele pode ser utilizado como registrador destino para uma operação cujo resultado deseja-se descartar ou como fonte para uma operação onde o valor zero é necessário [41].

Os módulos *Exception Unit* e *COP0* trabalham em conjunto no tratamento de exceções. Quando um sinal de exceção é gerado pelo módulo *Execute*, o módulo *Exception*

Unit lança a exceção através da configuração de registradores e, então, o módulo *Fetch* recebe o endereço de desvio para o tratamento de exceções. O módulo *COP0* salva o *status* do sistema para que a execução do programa retorne após o tratamento da exceção.

Segundo o manual MIPS Technologies, Inc [41], existem cinco condições definidas pelo padrão IEEE nas quais exceções são geradas:

- Operação inválida: quando o *opcode* da instrução não pertencente ao ISA
- Divisão por zero
- *Overflow*: quando a magnitude de um ponto flutuante arredondado excede a capacidade do local de destino
- *Underflow*: quando um número é muito pequeno a ponto de prejudicar sua representatividade
- Inexatidão: quando um resultado arredondado de uma operação não é exato

Ademais, o módulo *Router* realiza a função de barramento do sistema e controla as transações entre o modelo MIPS32 e os módulos de memória RAM, ROM e o canal UART (*Universal Asynchronous Receiver/Transmiter*). Já o módulo de *Debug* foi implementado com o intuito de validar o funcionamento do processador de uma forma não automatizada, possibilitando escrita e leitura em cada módulo do processador e imprimindo na tela de execução os resultados. Os *sockets* dos módulos do processador que são conectados ao módulo *Debug* são ativados mediante uma macro definida no momento de compilação, caso contrário eles não são criados.

6 Metodologia

O objetivo deste capítulo é descrever a metodologia aplicada para a construção do ambiente de verificação do processador MIPS32. Com base no que foi exposto no capítulo 3, pretende-se montar um plano de verificação fidedigno, explorando as estratégias de verificação e cumprindo os principais requisitos do processo de verificação, tentando aproximar o trabalho dos desafios que são, de fato, vivenciados no mercado.

Segundo Mishra e Dutt [43], a verificação de microprocessadores é uma das mais complexas e caras tarefas do atual processo de desenvolvimento de SoCs. Um gargalo significante na verificação destes sistemas é a falta de métricas de cobertura adequadas. Milhares de rodadas de simulação são gastas no fluxo comum de verificação e muitas métricas de cobertura são comumente usadas, tais como cobertura de código e cobertura de mudanças. Infelizmente, estas métricas não têm nenhuma relação direta com a funcionalidade do dispositivo. Por exemplo, nenhuma delas determina se todos *hazards* e exceções foram testadas na *pipeline* do processador. Portanto, há a necessidade de se criar métricas de cobertura baseadas na funcionalidade do modelo.

Como recomendado por Vasudevan [15] e citado na seção 3.1, foram coletadas várias informações pertinentes para a elaboração do plano de verificação. Primeiramente, as especificações de arquitetura foram descritas detalhadamente na subseção 5.4.2. Sabe-se que, por ser um sistema programável, este sistema eletrônico pode apresentar uma infinidade de comportamentos diferentes dependendo da lógica implementada no algoritmo do programa. Ainda assim, deve atender às especificações da folha de dados do fabricante.

Através da figura 22 já é conhecida a lista de entradas e saídas do processador, sendo que este está limitado pelo retângulo pontilhado. Os módulos externos não serão levados em conta neste trabalho, podendo ser acrescentados ao estudo de caso em trabalhos posteriores. Desta forma, o *testbench* deve-se encaixar às estas entradas e saídas de forma a simular o ambiente com o qual o processador trabalharia na realidade.

E, por fim, os cenários críticos nos quais o modelo pode apresentar funcionamento indevido são:

- Leitura e escrita de registradores
- Erro de decodificação da instrução
- Erro de execução da instrução
- Tratamento de exceções

- Falhas no desvio de programa

Tendo estas preciosas informações em mão, podemos responder a duas perguntas: *O que verificar? E como verificar?*

A primeira pergunta é relativamente simples e já foi parcialmente respondida. Pelo fato do trabalho estar focado na verificação funcional e pelo estudo de caso ser um processador, é necessário constatar e validar a implementação dos requisitos funcionais do modelo de acordo com o padrão de arquitetura descrito em seu manual. Deve-se, para isto, cobrir não só os cenários críticos do sistema, mas também as funcionalidades básicas por ele implementadas.

A segunda pergunta é de fato crucial. Sua resposta, por não ser retórica, será desenvolvida ao longo das próximas seções, onde o Plano de Verificação será construído.

6.1 Definição do Nível de Verificação

Como a implementação do MIPS32 trata-se de um modelo transacional de alto nível, contando basicamente com sete módulos instanciados em um *top level*, o nível de verificação pode variar entre o nível de *chip* e o de projetista (vide figura 8). Portanto, é possível verificar a arquitetura sob três óticas diferentes:

- Nível de *chip*: verificar o modelo neste nível, significa testar o processador como um todo, conectando o *testbench* às suas entradas e saídas sem infringir suas fronteiras.
- Nível de unidade: a verificação neste nível deve consistir em verificar a *pipeline* isolada dos demais módulos do processador, ou verificar o conjunto de módulos *Exception Unit* e *COP0*, ou ainda, verificar conjuntos de módulos em combinações diferentes.
- Nível de projetista: neste nível, a verificação deve ser feita com cada módulo do sistema isolado.

O ideal para um processador, é que ele seja verificado ao menos utilizando o nível de projetista e o nível de *chip*. O nível de projetista serviria para testar separadamente os módulos da *pipeline*, isolando falhas e evitando a propagação das mesmas quando forem executados testes em níveis superiores [7]. O nível de *chip* serviria para testar o conjunto completo do processador e verificar a comunicação e harmonia entre os módulos.

Esperava-se fazer a verificação do processador em ambos os níveis, contudo houve um inesperado gasto de tempo para a configuração do ambiente de trabalho, domínio da metodologia UVM e entendimento do modelo a ser testado. Isto já era de se esperar, pois o

processo de verificação é uma etapa que envolve uma equipe grande de engenheiros e muito planejamento prévio [4] [6]. Assim, houve uma mudança no cronograma de atividades e decidiu-se por efetuar a verificação completa do estágio *Decode* da *pipeline* no nível de projetista. Contudo, mesmo sendo um módulo pequeno, todos os principais artifícios disponíveis na metodologia UVM para construção de *testbenches* foram utilizados e podem ser expandidos para outros módulos e para níveis de verificação mais altos.

6.2 Definição das Métricas de Cobertura

O propósito das métricas de cobertura é medir o progresso dos testes realizados. Definindo boas metas de cobertura é possível executar uma verificação de qualidade e saber a hora exata de encerrar os testes. Isto minimiza o custo de projeto e também auxilia no cumprimento dos prazos de entrega.

Aqui serão definidas apenas as métricas de cobertura funcional que são as mais importantes e são viabilizadas pela utilização da metodologia UVM. Os valores estimados para alcançar as metas de cobertura foram escolhidos por vontade própria, pois não encontrou-se na literatura recomendações para estes valores. E, embora a cobertura de código não seja aborda neste trabalho, ela deve ser levada em conta e pode ser realizada com o auxílio de ferramentas do próprio simulador.

No estágio *Decode* é necessário verificar a leitura e escrita dos registradores, bem como a decodificação correta da instrução. As metas de cobertura deste estágio são definidas na tabela 3.

Critério de Cobertura	**Meta**
Escrita	10000 vezes cada registradores
Leitura	10000 vezes cada registrador como fonte e destino
Decodificação	1000 vezes cada *opcode* possível

Tabela 3: Métricas de cobertura funcional para o estágio *Decode*

Estas metas de cobertura dizem que cada registrador deve ser escrito 10000 vezes. Além disto, cada registrador deve ser lido também 10000 vezes, sendo 10000 vezes como registrador fonte (campo *rs* da palavra de instrução) e mais 10000 vezes como registrador alvo (campo *rt*). Os valores possíveis no campo de *opcode* da instrução devem ser atingidos pelo menos 1000 vezes cada um. Isto irá gerar um número de combinações muito grande até a cobertura ser atingida.

Como estes valores são estipulados de forma empírica, pode-se inclusive aumentá-los, visto que o processador real irá trabalhar com decodificação de áudio, o que requer um grande volume de cálculos durante longos períodos de tempo. Assim, efetuando uma maior quantidade de testes, a simulação irá se aproximar mais da situação em que o modelo irá trabalhar, como descrito na subseção 3.1.2.

6.3 Definição de Como Gerar Estímulos

Para alcançar as metas de cobertura com sucesso, o ambiente de verificação deve gerar um ciclo de testes escrevendo no banco de registradores e, posteriormente, enviando instruções a serem decodificadas, tal como mostra o fluxograma da figura 24.

Figura 24: Fluxograma para geração de estímulos

A escrita no banco de registradores sempre será feita de forma sequencial, isto é, do registrador *R0* ao registrador *R31*. Porém, os dados escritos em cada registrador são aleatórios, de forma com que cada registrador contenha um valor diferente a cada nova rodada de escrita. Como o estágio *Decode* não faz distinção do conteúdo dos campos da palavra de instrução (ele apenas os separa), o campo de 5 bits correspondente ao *opcode* também pode ser gerado de forma aleatória. Assim, o *testbench* deve permanecer gerando instruções aleatórias até que a meta de cobertura de leitura e decodificação sejam alcançadas.

Como descrito na seção 3.1.3, a geração de estímulos aleatórios é importante, pois como ela foge a um padrão de lógica, é possível criar cenários que não foram pensados no momento da construção do *testbench*. Isto aumenta a possibilidade de criar um cenário inesperado e atingir um caso crítico do sistema.

Vale lembrar que este módulo envia adiante todos os campos da instrução, incluindo os valores contidos nos campos de registradores *rs* e *rt*, de forma que a cada ciclo haja uma nova leitura.

6.4 Definição de Como Checar Respostas

É de suma importância validar os dados devolvidos pelo DUT ao ambiente de verificação. Para isto, o ambiente de verificação deve ter o controle exato do que está sendo enviado para o DUT como estímulo, para, posteriormente, efetuar a comparação entre estímulo e resposta. As informações referentes à quantidade de transações enviadas, acertos e erros do modelos devem ser impressas na tela de simulação. Caso erros sejam encontrados, o *testbench* deve ser capaz de indicá-los precisamente para facilitar o processo de depuração.

Este mecanismo de checagem é implementado através de monitores que coletam os dados devolvidos pelo DUT e enviam-nos para um Comparador. Este Comparador também recebe os dados de entrada e, internamente, possui uma função que executa a comparação dos dados de entrada e de saída baseada no modelo comportamental esperado descrito na folha de dados.

Este aspecto será melhor entendido no próximo capítulo, onde o ambiente de verificação construído será mostrado e explicado com detalhes. Os trechos de código e as imagens da arquitetura do *testbench* também irão contribuir para um melhor entendimento.

7 Resultados e Discussões

Tendo elaborado o plano de verificação, é hora de construir o *testbench*. Este capítulo será dedicado a mostrar como a metodologia descrita no capítulo 6 foi posta em prática por meio das ferramentas disponíveis. Os trechos mais importantes do código desenvolvido serão explicados e o código completo encontra-se em anexo, bem como o respectivo *makefile* necessário para a compilação do projeto.

7.1 Configuração do Ambiente de Trabalho

O material utilizado para o desenvolvimento deste trabalho foi:

- Biblioteca UVM-1.1d [29]
- Biblioteca UVM-2.2 *Connect* [30]
- Simulador Cadence Incisive versão 13.2
- Computador pessoal com processador Intel *core* i5, 8GB de memória RAM e sistema operacional CentOS 5.9

Tendo em mãos todo o material descrito acima, é necessário, primeiro, configurar o ambiente computacional para funcionar corretamente com as bibliotecas UVM e UVMC. Para isto, deve-se colocar os arquivos baixados em uma pasta de fácil acesso e descompactá-los. Feito isto, o próximo passo é criar as variáveis de ambiente UVM_HOME, UVMC_HOME e IUS_HOME. A primeira variável deve indicar o caminho da pasta raiz da biblioteca UVM-1.1d, a segunda variável deve indicar o caminho da pasta raiz da biblioteca UVMC-2.2 e a terceira variável deve indicar o caminho da pasta raiz do simulador Cadence Incisive. Pode-se usar, para isto, as seguintes linhas de comando:

$ setenv UVM_HOME /caminho/da/pasta/raiz/UVM
$ setenv UVMC_HOME /caminho/da/pasta/raiz/UVMC
$ setenv IUS_HOME /caminho/da/pasta/raiz/IUS

Os arquivos que compõem este projeto são:

- tb_main.sv (módulo *top* em SystemVerilog)
- testbench.sv (arquivo do ambiente de verificação)

- tb_main.cpp (módulo *top* em C++)
- decode.cpp (arquivo de implementação do DUT)
- decode.h (header do DUT)
- plasma_pack.h (header do processador)

Os arquivos tb_main.sv, tb_main.cpp e decode.cpp necessitam ficar na pasta raiz. Os demais arquivos devem ser colocados em uma subpasta chamada "common".

Por último, basta rodar de dentro da pasta raiz o *makefile* com a seguinte linha de comando:

$ makefile -f Makefile.ius testbench

O código completo do *makefile* pode ser conferido no anexo A. Ele foi baseado no próprio exemplo que acompanha a biblioteca UVMC, retirando-se a parte inutilizada e acrescentando duas linhas para habilitar as medidas de cobertura durante a simulação:

-coverage all
-covoverwrite

Ao executar o *makefile*, são criados arquivos de *log* e duas pastas geradas pelo simulador: INCA_libs e cov_work. É recomendado fazer uma limpeza destes arquivos antes de executar uma nova simulação, portanto, deve-se utilizar a linha de comando:

$ makefile -f Makefile.ius clean

A pasta cov_work não irá sumir com este comando. Para que isto aconteça, é necessário editar o arquivo Makefile.ius que fica na pasta /$(UVMC_HOME)/examples. A opção *clean* ficará assim:

*rm -rf * core csrc cov_work INCA* vc_hdrs.h ucli.key *.log*

7.2 Construção do *Testbench*

Primeiramente, é necessário entender a arquitetura proposta para o *testbench*. Tendo uma noção de disposição dos componentes e suas interligações, fica fácil entender o código criado. A figura 25 mostra a arquitetura completa do *testbench*, omitindo os módulos *top*, que servem apenas para instanciar o *testbench* e o módulo *Decode*.

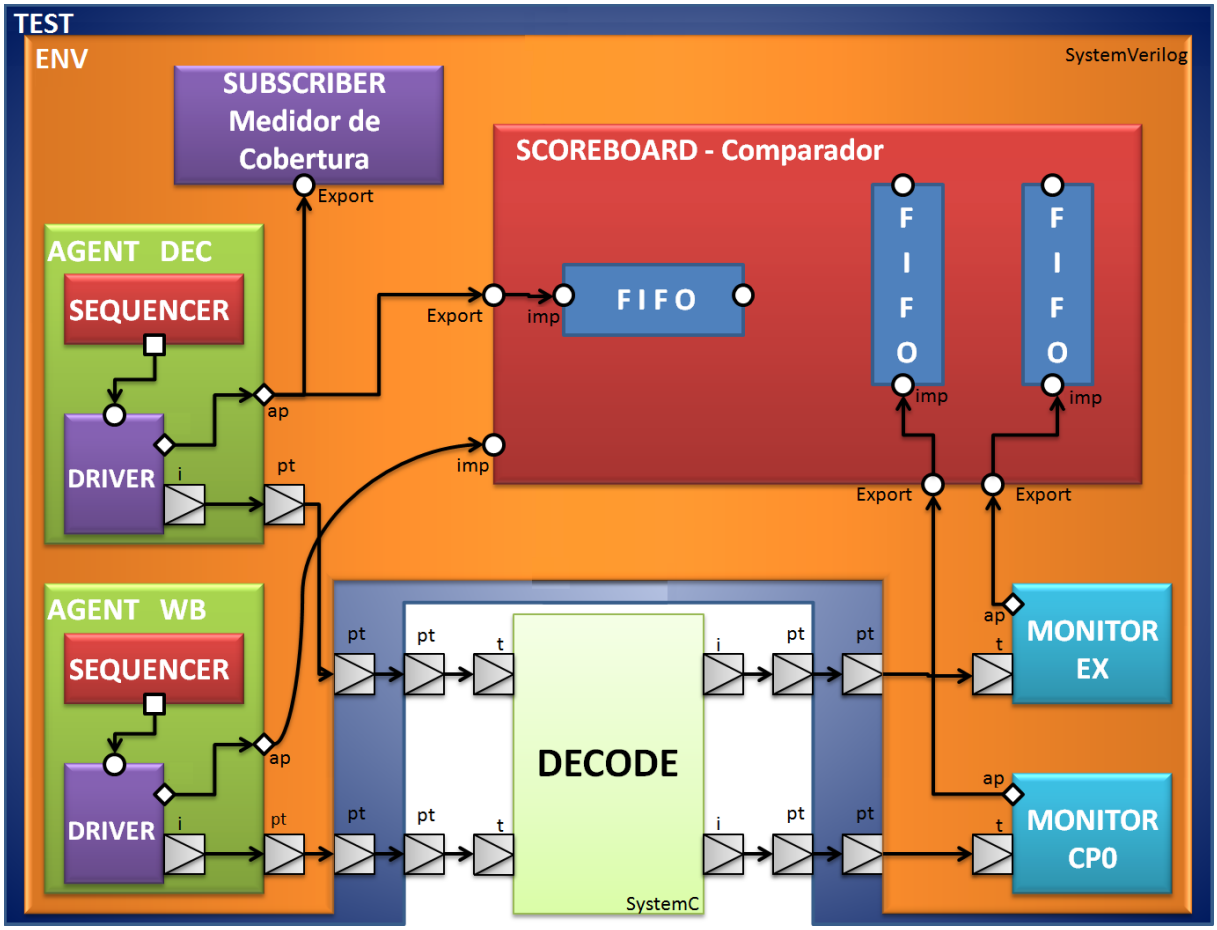

Figura 25: Arquitetura do *Testbench* Proposto

Na figura 25, é possível ver diversas estruturas da UVM e do TLM. O módulo *Decode* é a única porção de código em SystemC e todo o ambiente de verificação é construído em SystemVerilog. Os Agentes possuem em seu interior Sequenciadores e *Drivers* para gerar sequências de estímulos que serão enviados para o DUT. O *Agent Dec* é responsável por enviar sequências de instruções para serem decodificadas e o *Agent WB* é responsável por enviar sequências de dados para serem escritos no banco de registradores.

O *Subscriber* (Assinante) faz a medida de cobertura da simulação. Ele irá sinalizar quando todos os casos desejados forem alcançados, permitindo, assim, que a simulação termine. Os Monitores, por sua vez, coletam os dados de saída do DUT e enviam para o *Scoreboard* (Marcador) que fará a comparação dos dados de entrada com os dados de saída. Dentro do *Scoreboard* existem três estruturas chamadas de FIFO que servem para sincronizar a recepção dos dados, pois o primeiro dado a entrar é o primeiro também a sair.

Todos estes componentes estão instanciados dentro de um ambiente *Env* que, por sua vez, está instanciado em um *Test* e é através dele que a simulação é controlada.

Os losangos brancos representam portas de análise e servem para distribuir dados

a qualquer interface que esteja a ela conectada. Os círculos brancos podem representar *imps* ou *exports*. Os *sockets* marcados com a letra *i* são *initiators*, os marcados com a letra *t* são *targets* e os marcados com *pt* são *passthrough*. Este último tipo faz o papel de um componente de interconexão, apenas repassando o dado através da hierarquia sem modificá-lo.

As subseções seguintes explicam como instanciar essas estruturas e conectá-las em um *testbench*. Serão descritas particularidades de cada uma delas e também eventuais técnicas de programação que auxiliaram na construção do código. O código completo pode ser conferido no anexo D.

7.2.1 Agente e Monitor

Em geral, a estrutura de código de um componente de verificação da UVM segue um padrão muito parecido. Agentes e Monitores são componentes básicos de um ambiente de verificação. Embora os Monitores estejam normalmente contidos dentro dos próprios Agentes, aqui foi tomada a liberdade de instanciá-los direto no *Env*. O código 7.1 mostra um exemplo completo de como elaborar um *Driver*. Para todos os outros componentes, o processo de criação é similar. Observa-se que nem todas as fases citadas na seção 4.3 foram utilizadas e isto varia de componente para componente de acordo com a necessidade. As etapas para criação são:

1. Estender a classe base para adquirir herança (linha 1);
2. Registrar o componente com uma macro padrão da UVM (linha 2);
3. Instanciar portas de análise, *sockets* e atributos a serem utilizados no componente (linhas 4 e 5);
4. Declarar o construtor da classe (linhas 7 a 11);
5. Declarar a função que será chamada na fase *build_phase()* (linhas 13 a 15);
6. Declarar a *task* a ser executada na fase *run_phase()* (linhas 17 a 31);
7. Encerrar a classe (linha 32).

Código 7.1: Código completo do componente *Driver*

```
1 class my_driver extends uvm_driver #(uvm_tlm_generic_payload);
2   `uvm_component_utils(my_driver)
3
4   uvm_tlm_b_initiator_socket #() isocket;
5   uvm_analysis_port #(uvm_tlm_generic_payload) aport;
```

```
6
7    function new(string name, uvm_component parent=null);
8       super.new(name,parent);
9       isocket = new("isocket", this);
10      aport = new("aport", this);
11   endfunction
12
13   function void build_phase(uvm_phase phase);
14      super.build_phase(phase);
15   endfunction
16
17   task run_phase (uvm_phase phase);
18      uvm_tlm_generic_payload gp;
19      uvm_tlm_time delay = new("del",1e-9);
20
21      forever
22      begin
23         seq_item_port.get_next_item(gp);
24         delay.set_abstime(0,1e-9);
25         isocket.b_transport(gp, delay);
26         if(gp.is_response_error())
27            `uvm_info("DRIVER", $psprintf("ERROR -> Response status: %s\n\n",
                  gp.get_response_string) , UVM_LOW)
28         aport.write(gp);
29         seq_item_port.item_done();
30      end
31   endtask
32 endclass
```

A fase *run_phase()* é a única que consome tempo de simulação. Então, a funcionalidade do componente deve ser descrita dentro da respectiva *task*. O classe base *uvm_driver* possui uma porta chamada *seq_item_port* específica para se comunicar com um Sequenciador. É através dela que o *Driver* apanha um novo item da sequência invocando o método *get_next_item( )*.

Após pegar um uma nova transação (linha 23), o *Driver* a envia através do *initiator socket* chamando o método *b_transport()* (linha 25). O tempo é ajustado para zero, já que o modelo a ser verificado é atemporal (linha 24). Após enviar a transação pelo *socket* e verificar o sucesso da comunicação (linhas 26 e 27), a mesma transação é disponibilizada na porta de análise (linha 28). Finalmente, o *Driver* comunica ao Sequenciador que uma nova transação pode ser enviada (linha 29) e o ciclo se repete.

O sequenciador neste *testbench*, por não implementar nenhuma funcionalidade nova, dispensa a necessidade de estender a classe *uvm_sequencer*. Desta forma, um novo tipo de Sequenciador é criado como mostra o código 7.2.

Código 7.2: Como declarar um novo Sequenciador

```
1    typedef uvm_sequencer #(uvm_tlm_generic_payload) my_sequencer;
```

O Agente engloba um *Driver* e um Sequenciador, por isso ele deve instanciá-los, construí-los durante a *build_phase()* e, depois, conectá-los durante a *connect_phase()*. Esta última fase não estava presente no *Driver*, pois lá não havia nenhum componente a ser conectado. O código 7.3 mostra como isto é feito.

Código 7.3: Funções para construção e conexão no Agente

```
1   (...)
2   function void build_phase(uvm_phase phase);
3      super.build_phase(phase);
4      my_sequencer_h  = my_sequencer::type_id::create("my_sequencer_h", this);
5      my_driver_h     = my_driver::type_id::create("my_driver_h", this);
6   endfunction: build_phase
7
8   function void connect_phase(uvm_phase phase);
9      my_driver_h.seq_item_port.connect( my_sequencer_h.seq_item_export );
10     my_driver_h.isocket.connect( agent_pt_socket );
11     my_driver_h.aport.connect( aport );
12  endfunction
13  (...)
```

Na função *build_phase()*, o construtor da classe base da UVM é chamado na linha 3. Após isto, são construídos o Sequenciador e o *Driver* nas linhas 4 e 5, respectivamente. Lembrando que estes componentes devem ser anteriormente instanciados dentro da classe para que possam ser construídos. Na *connect_phase()* ocorrem três conexões: a *seq_item_port* do *Driver* é conectada à *seq_item_export* do Sequenciador (linha 9); o *passthrough socket* é conectado ao *socket* do *Driver* (linha 10); a porta de análise do *Driver* é conectada à porta de análise do Agente para que o dado seja disponibilizado ao ambiente de verificação (linha 11).

Por fim, o Monitor possui uma particularidade devido ao seu *target socket*. Ele precisa implementar uma *task* virtual chamada *b_transport()* que será executada toda vez que uma nova transação for iniciada. Ela é mostrada a seguir no código 7.4. Depois de receber a transação, é dada uma resposta positiva para o *initiator* (linha 2) e o dado coletado é disponibilizado na porta de análise para Marcador (linha 3).

Código 7.4: *Task* para receber transações através do *target socket*

```
1  virtual task b_transport (uvm_tlm_generic_payload t, uvm_tlm_time delay);
2      t.set_response_status(UVM_TLM_OK_RESPONSE);
3      aport.write(t);
4  endtask
```

7.2.2 Medidor de Cobertura

O medidor de cobertura constitui peça fundamental de um ambiente de verificação, pois é ele quem determina quando o teste chega ao fim. O SystemVerilog é uma linguagem poderosa de verificação e possui ferramentas específicas para tratar a cobertura. Através do chamado *covergroup*, é possível estabelecer metas de verificação no ambiente de verificação. Vejamos o trecho de código 7.5 pertencente ao componente *my_subscriber*.

Código 7.5: Grupo de Cobertura

```
1   bit [5:0]     op;
2   bit [4:0]     rs;
3   bit [4:0]     rt;
4
5   covergroup cover_group;
6         coverpoint op{
7               bins o[] = {[0:63]};
8               option.at_least = 1000; }
9         coverpoint rs{
10              bins s[] = {[0:31]};
11              option.at_least = 10000; }
12        coverpoint rt{
13              bins t[] = {[0:31]};
14              option.at_least = 10000; }
15  endgroup
```

As variáveis *op*, *rs* e *rt*, declaradas nas linhas de 1 a 3, são campos da palavra de instrução do processador e correspondem ao *opcode*, registrador fonte e registrador de destino, respectivamente. Dentro do *covergroup* é criando um ponto de cobertura específico para cada uma destas variáveis (linhas 6, 9 e 12). Pode-se então adicionar *bins* (linhas 7, 10 e 13), que é uma opção que divide os possíveis valores que a variável pode receber. No caso, todas as três variáveis foram separadas no número máximo suportado. Isto é, variáveis de 6 bits separadas em valores de 0 a 63 e variáveis de 5 bits separadas em valores de 0 a 31.

Acontece que, durante a simulação, cada vez que um valor destas variáveis for alcançado, o contador de cobertura é incrementado. A opção *option.at_least* (linhas 8, 11 e 14) insere o número mínimo de vezes que cada valor deve ser atingido. Como descrito na seção 6.2 do plano de verificação, definiu-se 10000 leituras para cada um dos registradores, como registrador destino e registrador fonte, e 1000 vezes cada valor possível do campo *opcode* das instruções.

Como esta classe herda uma *analysis_export*, ela deve implementar um método *write()* que possa ser executado quando uma transação estiver chegando. Então, neste método executa-se o mostrado no código 7.6.

Código 7.6: Método *write()* do medidor de cobertura

```
1  function void write(uvm_tlm_generic_payload t);
2        (...)
3        op = opcode[31:26];
4        rs = opcode[25:21];
5        rt = opcode[20:16];
6
7        cover_group.sample();
8        `uvm_info("SUBSCRIBER", $psprintf("Coverage: %0d%%", $get_coverage())
            ,UVM_LOW)
9        if($get_coverage() > 99)
10            coverage_achieved = 1;
11 endfunction
```

A primeira coisa a ser feita é resgatar os valores dos campos *op*, *rs* e *rt* da palavra de instrução (linhas 3 a 5). O método *sample()* do *covergroup* é executado e faz com que o valor da cobertura seja atualizado (linha 7). Posteriormente, na linha 8, uma mensagem é impressa na tela de simulação mostrando o valor em porcentagem obtido através do método *$get_coverage()*. Por fim, se o valor da cobertura for igual a 100, uma *flag* é levantada para indicar que o teste pode terminar (linhas 9 e 10).

7.2.3 Comparador

É no comparador que a validação comportamental do DUT acontece. O Comparador necessita receber todos os dados de entrada e de saída para compará-los segundo a descrição de requisitos comportamentais do sistema. Quando ocorre a escrita sequencial no banco de registradores, o Comparador também deve armazenar estes dados para que possa usá-los mais tarde. A porta de análise do *Agent WB* está ligada na porta *imp* do comparador de forma que os dados enviados para o DUT também sejam recebidos pelo Comparador. Por este motivo, um método *write()* deve ser implementado.

O restante dos dados é capturado através das interfaces *export* durante a *run_phase()* em um laço infinito e é repassado para as FIFOs. Isto é necessário para haver sincronismo na recepção, pois durante o tempo de simulação há uma grande atividade no ambiente de verificação e não se pode garantir que estes dados chegarão em ordem. Como mostra o trecho de código 7.7, o método *get()* chamado nas linhas de 1 a 3 para cada FIFO é bloqueante, ou seja, o código só retorna desta chamada quando uma transação é recebida.

Código 7.7: Captura de dados das interfaces FIFO e testes

```
1     fifo_in.get(gp_in);
2     fifo_ex.get(gp_ex);
3     fifo_cp0.get(gp_cp0);
4     erro = 0;
5
6     if(gp_in.m_length != 4)
7     begin
8        $sformat(s, "OPCODE Length does not match");
9        uvm_report_error("FIFO Error", s);
10       erro = 1;
11    end
12    if(gp_ex.m_length != 48)
13       (...)
14    if(gp_cp0.m_length != 32)
15       (...)
16    (...)
```

Existem três FIFOs: fifo_in, fifo_ex e fifo_cp0. A primeira recebe as instruções que são enviadas para o DUT, a segunda recebe as instruções decodificadas para o estágio *Execute* e a terceira recebe os dados de saída para o coprocessador COP0. Para cada FIFO existe uma *generic payload* correspondente e a primeira coisa a se fazer após a recepção é conferir se o tamanho de cada transação está correto através do campo m_length (linhas 6, 12 e 14).

Após este passo, os dados são transferidos para variáveis e então é feita uma série de testes para conferir a corretude do modelo. O testes são feitos com estruturas do tipo *if* e caso alguma discrepância seja encontrada, uma *flag* é utilizada para sinalizar a presença de erro. Logo ao final dos testes, as variáveis *m_match* e *m_mismatches* são utilizadas para computar o sucesso ou a falha do DUT.

7.2.4 Sequências de Transações

A Sequência de transações é responsável por randomizar os dados contidos na transação. Segundo a estrutura de classes da UVM, a Sequência não é um componente do

ambiente de verificação, ela é um objeto. Esta simples distinção permite que as Sequências sejam randomizadas e alteradas em tempo de simulação. O Sequenciador é utilizado para enviar a próxima transação ao *Driver* e serve apenas como um intermediador.

Neste ambiente de verificação, existem três sequências: uma que gera instruções, outra que gera dados de escrita para os registradores e uma Sequência Mestra, que controla a ação das outras duas sequências. Uma sequência que controla outras sequências é chamada de Sequência Virtual e deve ser utilizada quando o *testbench* possui múltiplos *Drivers* [16]. Desta forma, pode-se controlar a ordem de dados enviados para o DUT. Quando a sequência é iniciada, sua *task* principal entra em execução. O código 7.8 mostra o corpo da sequência de dados:

Código 7.8: Corpo da Sequência de dados

```
1  task body;
2   uvm_tlm_generic_payload gp;
3
4    for(int i = 0; i<32; i++)
5    begin
6       gp = uvm_tlm_generic_payload::type_id::create("gp");
7       start_item(gp);
8       assert( gp.randomize() with { gp.m_command == UVM_TLM_WRITE_COMMAND;
9                                     gp.m_length == 8;
10                                    gp.m_data.size() == gp.m_length;
11                                    gp.m_data[4] == i;
12                                    gp.m_data[5] == 0;
13                                    gp.m_data[6] == 0;
14                                    gp.m_data[7] == 0;
15                                    (...) } );
16      finish_item(gp);
17   end
18 endtask
```

Um novo item da sequência é iniciado com o método *start_item()* na linha 7. Depois, utiliza-se uma asserção para designar os valores que a transação irá receber (linha 8). Os dados a serem escritos são aleatório, porém a escrita nos registradores é sequencial. Assim, o método *randomize()* é restringido com o parâmetro *with*. As restrições desejadas são colocadas dentro deste campo. Neste caso, o campo m_data deve conter 8 bytes, sendo quatro deles para indicar o registrador e outros 4 contendo o dado a ser escrito. A atribuição do tamanho da transação se dá nas linhas 9 e 10.

É interessante notar neste ponto que o MIPS só possui 32 registradores, o que necessitaria apenas de 5 bits para serem endereçados. Porém, a construção em C++ não

permite esta restrição de bits que o SystemVerilog possui. No código do MIPS, o campo registrador é definido como um *unsigned int*, por isso contém 4 bytes.

Os campos 0, 1, 2 e 3 de m_data são correspondentes aos dados, por este motivo não devem conter restrição, caso contrário eles não sofrerão randomização. A variável de contagem *i* controla qual registrador será escrito e os demais bytes devem ser deixados como zero. Após definir os valores dos campos da transação o item é finalizado na linha 16 com o método *finish_item()*, indicando ao Sequenciador que esta transação já pode ser enviada ao *Driver*. Ao término da escrita nos 32 registradores, a sequência é automaticamente finalizada aguardando ser iniciada novamente pela Sequência Mestra.

7.2.5 Ambiente e Teste

O componente *Env* somente serve para instanciar e conectar todos os componentes de verificação. Como estes procedimentos já foram exemplificados anteriormente, não serão explicados aqui. O código completo pode ser conferido no anexo D, caso seja necessário.

O Teste é o componente de mais alto nível que compõe o ambiente de verificação. É através dele que as sequências são iniciadas para começar a estimular o DUT e é este componente que faz o controle de parada do testes. Ele deve observar a atividade do medidor de cobertura para saber qual é o momento de parar. O código 7.9 mostra a *task* da fase *run_phase()* deste componente.

Código 7.9: *Task* principal do componente Teste

```
1  task run_phase(uvm_phase phase);
2     master_sequence seq;
3     seq = master_sequence::type_id::create("seq");
4
5     phase.raise_objection(this);
6     seq.sequencer_dec = my_env_h.agent_dec.my_sequencer_h;
7     seq.sequencer_wb  = my_env_h.agent_wb.my_sequencer_h;
8     fork
9        begin
10          @(posedge my_env_h.subscriber_h.coverage_achieved);
11       end
12       begin
13          seq.start(null);
14       end
15    join_any
16    (...)
17    phase.drop_objection(this);
18 endtask
```

A primeira coisa que deve ser feita após instanciar e registrar a Sequência Mestra (linhas 2 e 3) é chamar o método *raise_objection()* (linha 5). Este método serve como um sinalizador indicando que há componentes ainda trabalhando na simulação. A *run_phase()* só termina quando a última objeção é abaixada. Assim, vários componentes podem levantar objeções, mas o manual da UVM [16] recomenda o controle de objeções somente do Teste.

Então, os Sequenciadores instanciados dentro da Sequência Mestra são associados com aqueles instanciados dentro dos Agentes (linhas 6 e 7) e ocorre uma bifurcação de processos com a chamada *fork()* na linha 8. O primeiro processo espera o fim do teste observando a borda de subida da *flag* coverage_achieved que pertence ao Assinante (medidor de cobertura). O segundo processo é o que dá início à Sequência Mestra através do método *start*. É a partir deste ponto que as outras duas sequências começam a operar criando dados que irão estimular o DUT. Vale lembrar que ambos os processos criados possuem execução simultânea.

A chamada *fork()* é finalizada com *join_any* na linha 15. Isto indica que a execução do processo pai irá continuar assim que qualquer um dos processos filho terminar. Como a *task* principal da Sequência Mestra consiste em um laço infinito, a chamada do método *start* nunca irá retornar. Assim, o processo pai irá parar a Sequência Mestra logo quando o Assinante sinalizar que as metas de cobertura foram alcançadas. Por fim, o Teste finaliza a *run_phase()* com a chamada de *drop_objection()* na linha 17. Quando a objeção é abaixada, a simulação segue para as fases seguintes. Neste *testbench*, a única fase posterior utilizada é a *report_phase()*, a qual imprime na tela os resultados obtidos durante a simulação.

7.2.6 Módulos *Top*

Os módulos *Top* servem apenas para instanciar o ambiente de verificação e o DUT. Deve haver um módulo *top* em SystemVerilog para o *testbench* e outro módulo *top* em C++ para o DUT. O código 7.10 apresenta o módulo sv_main que é construído em SystemVerilog e o código 7.11 apresenta o módulo sc_main que é construído em C++.

Código 7.10: Módulo *Top* em SystemVerilog

```
1 module sv_main;
2   my_test teste = new("teste");
3
4    initial begin
5       uvmc_tlm #()::connect(teste.test_dec_pt_socket, "fetch");
6       (...)
7       run_test();
8   end
```

Na porção SystemVerilog do código, o Teste é instanciando na linha 2 e, posteriormente, os *passthrough sockets* são conectados utilizando a biblioteca UVM *Connect*. Um exemplo do método *connect()* é mostrado na linha 5. Por fim, a simulação é iniciada através da chamada da função *run_test()* na linha 7.

A biblioteca UVMC só é utilizada nesta parte do código e conecta os *passthrough sockets* do Teste com os *sockets* do DUT, permitindo assim, a comunicação entre as diferentes linguagens. Observa-se também que as *strings* de registro devem ser idênticas em ambos os códigos, caso contrário haverá erro na fase de elaboração.

Código 7.11: Módulo *Top* em C++

```
1 int sc_main(int argc, char* argv[]){
2   Decode dec("dec");
3
4   uvmc_connect(dec.f_tsocket,"fetch");
5   (...)
6  sc_start(-1);
7   return 0;
8 }
```

Na porção C++ do código, o DUT é instanciado na linha 2 e a função *uvmc_connect()* é chamada na linha 4 para conectar o *socket* com o lado SystemVerilog do *testbench*. Por fim, a simulação é iniciada com a chamada da função *sc_start()* na linha 6.

7.3 Resultado da Simulação

Nesta seção, serão mostrados os resultados da simulação executada. Isto é, basicamente, as informações geradas pelo simulador antes, durante e após a simulação. Os arquivos de saída completos podem ser consultados nos anexos E e F.

É importante mostrar a ação da biblioteca UVM *Connect* no momento da elaboração, registrando os *sockets* e conectando-os. A seguir, é mostrado um trecho reduzido do resultado da simulação onde pode-se observar isto.

Listagem 7.12: Mensagens da UVMC no momento da elaboração

```
1 Connecting an SC-side proxy port for ‘dec.f_tsocket’ with lookup string ‘fetch’
     for later connection with SV
2 Connecting an SC-side proxy port for ‘dec.wb_tsocket’ with lookup string
    ‘writeback’ for later connection with SV
3 (...)
4 Registering SV-side ‘teste.test_dec_pt_socket’ and lookup string ‘fetch’ for
    later connection with SC
```

```
5 Registering SV-side ‘teste.test_wb_pt_socket’ and lookup string ‘writeback’ for
     later connection with SC
6 (...)
7 Connected SC-side ‘dec.e_isocket’ to SV-side ‘teste.test_ex_pt_socket’
8 Connected SC-side ‘dec.wb_tsocket’ to SV-side ‘teste.test_wb_pt_socket’
```

Primeiro, os *sockets* do lado C++ são registrados com a *string* informada no código (linhas 1 e 2). Depois, os *passthrough sockets* do lado SystemVerilog são registrados da mesma maneira (linhas 4 e 5). Por último, estes *sockets* são conectados para dar início à simulação (linhas 7 e 8).

No início da *run_phase()* os componentes do ambiente de verificação escrevem na tela de simulação a mensagem "*Starting...*" para demonstrar sua atividade e isto pode ser visto nas linhas 1 e 2 do código 7.13. Após isto, o Teste inicia a Sequência Mestra e o medidor de cobertura (*Subscriver*) começa a imprimir na tela a medida de cobertura em porcentagem (linhas 4 a 9). Quando a cobertura atinge 100%, o Teste encerra a simulação abaixando a objeção (linha 11).

Listagem 7.13: Mensagens do *testbench* durante a simulação

```
1 UVM_INFO ./common/testbench.sv(693) @ 0: teste [TEST] Starting...
2 UVM_INFO ./common/testbench.sv(202) @ 0: teste.my_env_h.monitor_ex [MONITOR]
    Starting...
3 (...)
4 UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 0%
5 (...)
6 UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 50%
7 (...)
8 UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 99%
9 UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 100%
10 UVM_INFO ./common/testbench.sv(716) @ 0: teste [TEST] Coverage Goals have been
    achieved!
11 UVM_INFO ./common/testbench.sv(717) @ 0: teste [TEST] Ending test and dropping
    objection...
12 UVM_INFO /home/alunos/junior_tcc2/uvm-1.1d/src/base/uvm_objection.svh(1268) @
    0: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract'
    phase
13 UVM_INFO ./common/testbench.sv(571) @ 0: teste.my_env_h.scoreboard_h [
    SCOREBOARD] Matches: 328238
```

```
14 UVM_INFO ./common/testbench.sv(572) @ 0: teste.my_env_h.scoreboard_h [
    SCOREBOARD] Mismatches: 0
15 UVM_INFO ./common/testbench.sv(281) @ 0: teste.my_env_h.subscriber_h [COVERAGE
    COLLECTOR] Number of Transactions: 328238
```

Após o Teste abaixar a objeção, entra em execução a *report_phase()*, que é utilizada para imprimir na tela resultados pertinentes obtidos durante a simulação. Observa-se na linha 15 que para atingir a cobertura foram necessárias 328238 transações. Destas transações enviadas, em nenhuma o DUT apresentou comportamento irregular (linhas 13 e 14). Este teste demorou em torno de 5 minutos para ser concluído e foi cronometrado de forma superficial com um relógio.

As informações são impressas na tela de acordo com o seu nível de importância. Para definir o nível de importância de uma mensagem, utiliza-se as macros UVM_LOW, UVM_MEDIUM e UVM_HIGH. No momento da compilação, deve-se ajustar também a macro UVM_VERBOSITY que está contida no *makefile*. Se esta macro for ajustada para LOW, somente as mensagens contendo a macro UVM_LOW serão impressas; se o ajuste for para MEDIUM, as mensagens contendo UVM_LOW e UVM_MEDIUM serão impressas; se o ajuste for para HIGH, todas as mensagens serão impressas. Por padrão, a macro UVM_MEDIUM é atribuída para as mensagens do tipo *uvm_report_info()* e a macro UVM_LOW para as mensagens *uvm_report_error()*. Este artifício possibilita inserir várias mensagens de depuração no código e filtrá-las no momento da simulação.

Após a simulação terminar, a UVM imprime uma série de dados sobre a simulação, tais como a quantidade de erros, advertências, sumário de mensagens enviadas, etc. Tudo isto pode ser conferido nos anexos.

Por fim, para verificar se o *testbench* estava realmente tratando os dados da forma correta, embutiu-se um erro no código do DUT na parte em que é feita a separação do campo *rd* (registrador de destino) da palavra de instrução. Um pequeno trecho é listado a seguir, no qual o Comparador, durante a simulação, imprime na tela mensagens indicando que houve uma discrepância entre os dados de entrada com os de saída.

Listagem 7.14: Mensagens do *testbench* sinalizando erros

```
1 (...)
2 UVM_INFO ./common/testbench.sv(312) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 16%
3 UVM_ERROR @ 0: teste.my_env_h.scoreboard_h [Comparator Decode Mismatch] RD does
     not match
4 UVM_ERROR @ 0: teste.my_env_h.scoreboard_h [Comparator COP0 Mismatch: ] RD does
     not match
5 (...)
```

```
6 UVM_INFO @ 0: teste.my_env_h.scoreboard_h [SCOREBOARD] Matches: 240
7 UVM_INFO @ 0: teste.my_env_h.scoreboard_h [SCOREBOARD] Mismatches: 7546
8 UVM_INFO @ 0: teste.my_env_h.subscriber_h [SUBSCRIBER] Number of Transactions
    collected: 7786
```

Este teste foi feito com uma meta de cobertura menor com o intuito apenas de economizar tempo. Observa-se que das 7786 transações enviadas, apenas 240 foram tratadas corretamente pelo DUT (linhas 6 a 8). A vantagem deste *testbench* é que ele indica exatamente em qual campo da palavra de instrução está havendo erros, o que facilita depuração de erros. Existem duas mensagens de erro, pois uma é referente aos dados que vieram da saída para o estágio *Execute* (linha 3) e a outra é referente aos dados que vão para o coprocessador COP0 (linha 4).

7.4 Trabalhos Futuros

Houveram muitas dificuldades ao longo da execução deste trabalho, a começar pela preparação do ambiente de trabalho. Como a licença para a utilização da ferramenta Cadence encontra-se em um servidor localizado em um *campus* diferente, foi necessário efetuar diversas modificações no terminal de trabalho para que as bibliotecas UVM e UVMC fossem compiladas. Esta mudança é local, ou seja, se outro computador fosse utilizado, a modificação deveria ser feita novamente.

Outros empecilhos foram enfrentados, mas, sem dúvida, a maior barreira encontrada foi a falta de documentação do modelo verificado. Como o trabalho de modelagem do processador ainda está em desenvolvimento, não existe uma documentação oficial que descreva seu funcionamento e estrutura interna a qual poderia ser consultada. Isto gerou a necessidade de entender o código do processador a fundo, o que não foi tão trivial. Isto encaixou-se na abordagem *white-box*, porém, há a certeza de que este processador pode ser verificado segundo a abordagem *black-box*, pois conhecendo-se bem as estruturas de dados que devem ser enviadas ou esperadas em cada entrada e saída do modelo, é possível a construção de um ambiente de verificação mais genérico e reutilizável.

Duas grandes críticas podem ser feitas a respeito do trabalho desenvolvido. A primeira é que não foi feita a verificação completa do processador, incluindo todos os seus estágios da *pipeline*, a unidade de tratamento de exceções e o coprocessador. Espera-se que este trabalho possa ter continuidade para que o objetivo principal seja concluído e para que o processador possa ser validado. A segunda crítica é que o critério de escritas nos registradores não foi incluído no Medidor de Cobertura. Do modo como o *testbench* encontra-se, os dados escritos nos registradores são renovados durante a simulação, mas não são mensurados pelo Medidor de Cobertura. Como estes dados vêm de um Agente diferente, deve-se criar um mecanismo para que o Medidor de Cobertura consiga receber

e tratar informações de todos os Agentes ativos do ambiente de verificação. Quando isto for feito, será possível também incluir nas metas de cobertura o critério de escrita nos registradores.

Uma técnica disponível na UVM que pode ser utilizada para melhorar o desempenho do *testbench* é a base de dados. Este recurso possibilita a construção de uma tabela com diversos parâmetros que configuram o *testbench*. Assim, parâmetros importantes da simulação podem ser controlados do Teste, tornando o *Env* e seus componentes, um bloco monolítico que não precisa ser alterado. Com isto, o engenheiro de verificação pode ter vários Testes escritos e ele só precisa indicar qual será usado no momento da compilação através de linha de código ou do *makefile*. Assim, mudando-se o teste, muda-se o foco da simulação e da verificação.

A geração de estímulos também pode ser melhorada buscando-se formas de alterar o tipo de distribuição dos elementos aleatórios e efetuando-se testes com sementes diferentes. Quando um teste é executado novamente sem ser feita nenhuma alteração, ele reenvia os mesmos dados na mesma ordem. Para que os dados mudem, deve-se trocar a semente, pois o algoritmo de geração dos estímulos é pseudo-aleatório. Alterar a tipo distribuição também é um recurso que pode auxiliar a criar cenários diferentes na simulação.

Outro melhoramento que pode ser feito no código é adequar a cópia entre tipos de dados diferentes. Por exemplo, o campo m_data da *generic payload* é um vetor do tipo byte e as variáveis utilizadas no código são um vetor do tipo bit. Isto causou erros na hora de copiar os dados de forma direta, necessitando fazer uma cópia individual de cada byte para que houvesse êxito. Esta solução provavelmente tornaria-se inviável caso o número de bytes a serem copiados fosse muito grande.

Este é um problema particular de adequação das linguagens. O C++ não trabalha com os tipos byte ou bit. Se uma variável booleana é utilizada, ela ocupará 4 bytes de qualquer forma. Estes testes foram feitos na fase de desenvolvimento para entender e contornar problemas encontrados. Assim, um possível melhoramento para a construção de *testbenches* que se comunicam com modelos em C++, seria a criação de uma nova classe que represente uma *generic payload* e que implemente os mesmo métodos, mas faça a adequação dos tipos de dados. Isto seria relativamente fácil de se fazer, já que o código da UVM é *open source*.

8 Conclusão

Neste trabalho foi apresentada a construção de um ambiente de verificação funcional utilizando a metodologia UVM. Os objetivos eram: buscar uma metodologia de verificação condizente com o praticado no mercado; aplicar esta metodologia a um modelo de processador descrito em nível transacional com SystemC; descrever um plano de verificação completo e específico para o estudo de caso; construir o ambiente de verificação atendendo às especificações do plano de verificação; realizar simulações com o ambiente de verificação e averiguar o funcionamento correto do processador.

Embora o objetivo maior deste trabalho não tenha sido alcançado, que consistia na completa verificação do processador, os resultados do trabalho tornaram possível entender o que é a verificação funcional e como ela é feita de uma forma profunda. A construção de um *testbench* aplicando a metodologia UVM foi realizada e, além disto, abre novos caminhos para uma linha de pesquisa que é muito pouco explorada no Brasil, tendo como maior referência a Universidade de Campina Grande, na Paraíba, com os trabalhos de Silva [44], Rodrigues [14] e Oliveira [25], dentre outros.

Na primeira parte deste trabalho, a revisão bibliográfica revelou duas tendências de mercado, sendo uma no campo de modelagem de sistemas e outra no campo de verificação funcional. Em se tratando de modelagem de sistemas, observou-se a incorporação de metodologias que guiassem e permitissem a modelagem em um alto nível de abstração, *e. g. ESL e TLM*. No campo da verificação, observou-se a união de diversos segmentos da indústria eletrônica para fortalecer o processo de verificação de seus produtos segundo um padrão aceito por todos. Então, a UVM se mostra fruto destas duas tendências. Além de ser um padrão aceito pela maioria dos líderes de mercado, possibilita a verificação de sistemas eletrônicos descritos tanto em baixo nível como também em alto nível. Isto é constatado principalmente pela incorporação de uma poderosa linguagem de verificação, o SystemVerilog, e a incorporação do padrão TLM-2.0, permitindo ainda, a interoperabilidade entre diversas linguagens diferentes.

Com a elaboração do Plano de Verificação, buscou-se atingir os pontos críticos do modelo estabelecendo-se metas de coberturas condizentes com suas funcionalidades. Isto, então, foi transposto para um ambiente de verificação utilizando os recursos da UVM e seguindo as principais diretrizes de construção recomendadas por Accellera [16], El-Metwally [19] e Jain et al. [45]. Várias rodadas de testes com critérios de cobertura diferentes foram feitas e constatou-se a corretude e bom funcionamento do módulo *Decode*.

As principais contribuições foram, primeiramente, a geração de um documento focado na metodologia UVM escrito em português e também a implementação de um

testbench específico para tratar modelos em nível transacional utilizando a UVM *Connect*. A revisão bibliográfica não levantou nenhum trabalho trazendo a abordagem UVM aplicada a SystemC e isto enfatiza a importância deste trabalho. A arquitetura adotada para o *testbench* pode servir de base e até mesmo ser aproveitada para verificar outros componentes do mesmo processador, reduzindo custos e esforços. Além disto, o código é livre para ser utilizado em outros trabalhos com o mesmo enfoque.

O trabalho também contribuiu a nível pessoal para aprender conceitos não adquiridos durante a graduação. Além dos conceitos de verificação funcional e de modelagem de sistemas que são vastos e possuem muitas abordagens, foi necessário aprender Orientação a Objetos para, então, aprender SystemVerilog, C++ e SystemC.

As limitações observadas durante a fase de implementação são relacionadas à diferença dos tipos de dados existentes entre SystemVerilog e C++. Isto retirou um pouco o caráter de reutilização do *testbench* e o atou muito ao modelo do estudo de caso.

Por fim, sugere-se a adequação e expansão do *testbench* para que possa ser feita a verificação do processador completo. A modificação proposta na seção 7.4 quanto à criação de uma nova classe de transação é uma boa solução para otimizar a escrita do código. Ademais, com as metodologias descritas neste trabalho, pode-se partir para a criação de VIPs (*Verirication Intelectual Property*) que podem ser reutilizados em diversos outros projetos.

Referências

1 TOBAR, E. L. R. *Contribuição à Verificação Funcional Ajustada Por Cobertura Para Núcleos de Hardware de Comunicação e Multimídia*. Tese (Doutorado) — Escola Politécnica, Universidade de São Paulo, 2010. Citado na página 25.

2 SANGIOVANNI-VINCENTELLI, A. L.; MCGEER, P. C.; SALDANHA, A. Verification of electronic systems. In: *Anais da 33ª Conferência Anual de Automação de Design*. New York, NY, USA: ACM, 1996. (DAC '96), p. 106–111. ISBN 0-89791-779-0. Citado na página 25.

3 ROWSON, J. A.; SANGIOVANNI-VINCENTELLI, A. Interface-based design. In: *Anais da 34ª Conferência Anual de Automação de Design*. New York, NY, USA: ACM, 1997. (DAC '97), p. 178–183. ISBN 0-89791-920-3. Citado na página 25.

4 MOLINA, A.; CADENAS, O. Functional verification: Approaches and challenges. *Latin American Applied Research*, scieloar, v. 37, p. 65 – 69, 01 2007. ISSN 0327-0793. Citado 4 vezes nas páginas 25, 26, 31 e 77.

5 BERMAN, V. Ieee p1647 and p1800: Two approaches to standardization and language design. *Design Test of Computers, IEEE*, v. 22, n. 3, p. 283–285, 2005. ISSN 0740-7475. Citado na página 25.

6 PAGLIARINI, S. N. *VEasy: A Tool Suite Towards the Functional Verification Challenges*. Dissertação (Mestrado) — Instituto de Informática, PGMicro, Universidade Federal do Rio Grande do Sul, 2011. Citado 8 vezes nas páginas 25, 26, 27, 35, 47, 48, 49 e 77.

7 BERGERON, J. *Writing Testbenches: Functional Verification of HDL Models*. [S.l.]: Kluwer Academic Publishers, 2003. ISBN 9781402074011. Citado 8 vezes nas páginas 29, 30, 31, 32, 33, 40, 42 e 76.

8 PIZIALI, A. *Functional Verification Coverage Measurement and Analysis*. [S.l.]: Springer, 2004. ISBN 9781402080258. Citado 11 vezes nas páginas 29, 33, 34, 35, 36, 39, 40, 42, 43, 44 e 45.

9 IEEE. *Standard VHDL Language Reference Manual*. [S.l.], 2009, 626 p. Citado na página 29.

10 IEEE. *IEEE Standard for Verilog Hardware Description Language*. [S.l.], 2006, 560 p. Citado na página 29.

11 IEEE. *SystemVerilog - Unified Hardware Design, Specification, and Verification Language*. [S.l.], 2009, 1285 p. Citado 2 vezes nas páginas 29 e 50.

12 IEEE. *Standard for Standard SystemC Language Reference Manual*. [S.l.], 2012, 638 p. Citado 2 vezes nas páginas 29 e 68.

13 COLLINS, J. D.; TULLSEN, D. M.; WANG, H. Control flow optimization via dynamic reconvergence prediction. In: *IN MICRO-37*. [S.l.: s.n.], 2004. p. 129–140. Citado na página 30.

14 RODRIGUES, C. L. *Análise de Cobertura Funcional na Fase de Integração de Blocos de Circuitos Digitais*. Tese (Doutorado) — Centro de Engenharia Elétrica e Informática, Universidade Federal de Campina Grande, 2010. Citado 4 vezes nas páginas 32, 33, 36 e 99.

15 VASUDEVAN, S. *Effective Functional Verification: Principles and Processes*. [S.l.]: Springer, 2006. ISBN 9780387326207. Citado 5 vezes nas páginas 32, 39, 42, 45 e 75.

16 ACCELLERA. *Universal Verification Methodology (UVM) 1.1 User's Guide*. Califórnia, Estados Unidos da América, 2011, 198 p. Citado 5 vezes nas páginas 36, 37, 90, 92 e 99.

17 MEYER, A. *Principles of Functional Verification*. [S.l.]: Elsevier Science, 2003. ISBN 9780080469942. Citado na página 39.

18 WILE, B.; GOSS, J.; ROESNER, W. *Comprehensive Functional Verification: The Complete Industry Cycle*. [S.l.]: Elsevier Science, 2005. (Systems on Silicon). ISBN 9780080476643. Citado 3 vezes nas páginas 40, 41 e 45.

19 EL-METWALLY, M. Guidelines for successful soc verification in ovm/uvm. Design and Reuse, Mentor Graphics Corp., 2011. Citado 2 vezes nas páginas 43 e 99.

20 PIZIALI, A.; CADENCE. Verification planning to functional closure of processor-based socs. *Incisive Newsletter*, Cadence Design Systems, 2006. Citado na página 44.

21 WEISZFLOG, W. *Michaelis Moderno Dicionário da Língua Portuguesa*. 2009. <http://michaelis.uol.com.br/moderno/portugues/index.php>. Acessado em: 14/05/2014. Citado na página 45.

22 GRIES, H. *Verification Methodology Poll Results*. 2009. <http://theasicguy.com/2009/02/11/verification-methodology-poll-results>. Acessado em: 16/05/2014. Citado 2 vezes nas páginas 47 e 48.

23 NANGIA, R. *Need of Verification Methodologies: Does SV Not Suffice?* 2014. <http://www.xinoe.com/blog/need-verification-methodologies-sv-suffice.html>. Acessado em: 16/05/2014. Citado na página 48.

24 VERISITY DESIGN. *e Reuse Methodology (eRM) Developer Manual*. [S.l.], 2002, 330 p. Citado 2 vezes nas páginas 48 e 49.

25 OLIVEIRA, H. F. d. A. *BVM: Reformulação da Metodologia de Verificação Funcional VeriSC*. Dissertação (Mestrado) — Centro de Engenharia Elétrica e Informática, Universidade Federal de Campina Grande, 2010. Citado 2 vezes nas páginas 50 e 99.

26 ANDERSON, T. et al. *SystemVerilog reference verification methodology: RTL*. 2006. <http://www.embedded.com/print/4004083>. Acessado em: 17/05/2014. Citado 2 vezes nas páginas 51 e 52.

27 DOULOS. *VMM Golden Reference Guide*: A concise guide to vmm verification methodology version 1.2. [S.l.], 2010, 364 p. Citado na página 52.

28 CADENCE E MENTOR GRAPHICS. *Open Verification Methodology User Guide*. [S.l.], 2011, 164 p. Citado 2 vezes nas páginas 53 e 54.

29 ACCELLERA. <http://www.accellera.org/downloads/standards/uvm>. Acessado em: 10/06/2014. Citado 2 vezes nas páginas 56 e 81.

30 GRAPHICS, M. <https://verificationacademy.com/resource/26993>. Acessado em: 10/06/2014. Citado 2 vezes nas páginas 56 e 81.

31 GAJSKI, D. et al. *Embedded System Design: Modeling, Synthesis and Verification*. [S.l.]: Springer, 2009. ISBN 9781441905048. Citado 2 vezes nas páginas 59 e 60.

32 BEIERLEIN, T.; HAGENBRUCH, O. *Taschenbuch Mikroprozessortechnik*. [S.l.]: Fachbuchverl. Leipzig im Carl-Hanser-Verlag, 2004. ISBN 9783446220720. Citado na página 61.

33 MARTIN, G.; BAILEY, B.; PIZIALI, A. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. [S.l.]: Elsevier Science, 2010. (Systems on Silicon). ISBN 9780080488837. Citado 3 vezes nas páginas 61, 62 e 63.

34 RIGO, S.; AZEVEDO, R.; SANTOS, L. *Electronic System Level Design: An Open-Source Approach*. [S.l.]: Springer, 2011. ISBN 9781402099403. Citado 2 vezes nas páginas 62 e 65.

35 BLACK, D. C. et al. *SystemC: From the Ground Up*. [S.l.]: Springer, 2009. (NATO ASI series: Cell biology). ISBN 9780387699585. Citado 4 vezes nas páginas 64, 65, 66 e 68.

36 AYNSLEY, J. *OSCI TLM-2.0 User Manual*. [S.l.], 2008, 151 p. Citado 3 vezes nas páginas 64, 65 e 70.

37 BESERRA, G. S. *Modelagem em Nível Transacional de Sistemas em Chip Mistos para Aplicações de Redes de Sensores sem Fio*. Tese (Doutorado) — Faculdade de Tecnologia, Universidade de Brasília, 2010. Citado 4 vezes nas páginas 66, 68, 69 e 70.

38 MONTOREANO, M. *Transaction Level Modeling using OSCI TLM 2.0*. [S.l.], 2007, 5 p. Citado 2 vezes nas páginas 67 e 68.

39 GRÖTKER, T. et al. *System Design with SystemC^{TM}*. [S.l.]: Springer, 2002. ISBN 9781402070723. Citado na página 68.

40 BAILEY, B. *The Functional Verification of Electronic Systems: An Overview from Various Points of View*. [S.l.]: International Engineering Consortium, 2005. (Design Handbook Series). ISBN 9781931695312. Citado na página 69.

41 MIPS TECHNOLOGIES, INC. *MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture*. 955 East Arques Avenue, Sunnyvale, CA 94085-4521, 2013, 116 p. Citado 4 vezes nas páginas 71, 72, 73 e 74.

42 HENNESSY, J.; PATTERSON, D.; ASANOVIĆ, K. *Computer Architecture: A Quantitative Approach*. [S.l.]: Morgan Kaufmann/Elsevier, 2012. (Computer Architecture: A Quantitative Approach). ISBN 9780123838728. Citado na página 71.

43 MISHRA, P.; DUTT, N. D. Functional coverage driven test generation for validation of pipelined processors. In: . [S.l.]: IEEE Computer Society, 2005. p. 678–683. ISBN 0-7695-2288-2. Citado na página 75.

44 SILVA, K. R. G. d. *Uma Metodologia de Verificação Funcional para Circuitos Digitais*. Tese (Doutorado) — Centro de Engenharia Elétrica e Informática, Universidade Federal de Campina Grande, 2007. Citado na página 99.

45 JAIN, A. et al. Early Development of UVM based Verification Environment of Image Signal Processing Designs using TLM Reference Model of RTL. *International Journal of Advanced Computer Science and Applications*, v. 5, 2014. Citado na página 99.

Anexos

ANEXO A – Arquivo *Makefile*

```
include $(UVMC_HOME)/examples/Makefile.ius

testbench:
   $(MAKE) -f Makefile.ius run EXAMPLE=tb_main

run:
   $(IUS) \
   -sc_main \
   -coverage all \
   -covoverwrite \
   -I$(UVMC_HOME)/src/connect/sc \
   -I./common \
   -DSC_INCLUDE_DYNAMIC_PROCESSES \
   -incdir ./common \
   $(EXAMPLE).cpp decode.cpp \
   $(EXAMPLE).sv \
   +UVM_VERBOSITY=MEDIUM \
   $(ARGS) \
      2>&1 | tee $(EXAMPLE).log
   $(CHECK)
```

ANEXO B – Código do Módulo *Top* SystemVerilog

```
import uvm_pkg::*;
import uvmc_pkg::*;
`include "testbench.sv"

module sv_main;
  my_test teste = new("teste");

   initial begin
      uvmc_tlm #()::connect(teste.test_dec_pt_socket, "fetch");
      uvmc_tlm #()::connect(teste.test_wb_pt_socket, "writeback");
      uvmc_tlm #()::connect(teste.test_ex_pt_socket, "execute");
      uvmc_tlm #()::connect(teste.test_cp0_pt_socket, "cop0");

      run_test();
  end
endmodule
```

ANEXO C – Código do Módulo *Top* C++

```
#include "uvmc.h"
#include "decode.h"

using namespace uvmc;

int sc_main(int argc, char* argv[])
{
   Decode dec("dec");

   uvmc_connect(dec.f_tsocket,"fetch");
   uvmc_connect(dec.wb_tsocket,"writeback");
   uvmc_connect(dec.e_isocket,"execute");
   uvmc_connect(dec.cp0_isocket,"cop0");

  sc_start(-1);
   return 0;
}
```

ANEXO D – Código do *Testbench*

```
import uvm_pkg::*;
`include "uvm_macros.svh"

//#############################################################################
class opcode_sequence extends uvm_sequence #(uvm_tlm_generic_payload);
   `uvm_object_utils(opcode_sequence)

   rand int n;
   constraint quantity { n inside {[32:70]}; }

   function new(string name = "");
      super.new(name);
   endfunction

   task body;
    uvm_tlm_generic_payload gp;

      `uvm_info("OPCODE SEQUENCE", $psprintf("Numero de instrucoes a serem
          enviadas: %0d", n) ,UVM_HIGH)
      repeat(n)
      begin
         gp = uvm_tlm_generic_payload::type_id::create("gp");
         start_item(gp);
         assert(gp.randomize() with{ gp.m_command == UVM_TLM_WRITE_COMMAND;
                                     gp.m_address == 0;
                                     gp.m_length == 4;
                                     gp.m_data.size() == gp.m_length;
                                     gp.m_streaming_width == gp.m_length;
                                     gp.m_byte_enable.size() == 0;
                                     gp.m_byte_enable_length == 0;});
         finish_item(gp);
      end
   endtask
endclass

//#############################################################################
class data_sequence extends uvm_sequence #(uvm_tlm_generic_payload);
   `uvm_object_utils(data_sequence)
```

```
   function new(string name = "");
      super.new(name);
   endfunction

   task body;
    uvm_tlm_generic_payload gp;

      for(int i = 0; i<32; i++)
      begin
         gp = uvm_tlm_generic_payload::type_id::create("gp");
         start_item(gp);
         assert(gp.randomize() with{ gp.m_command == UVM_TLM_WRITE_COMMAND;
                                     gp.m_address == 0;
                                     gp.m_length == 8;
                                     gp.m_data.size() == gp.m_length;
                                     gp.m_data[4] == i;
                                     gp.m_data[5] == 0;
                                     gp.m_data[6] == 0;
                                     gp.m_data[7] == 0;
                                     gp.m_streaming_width == gp.m_length;
                                     gp.m_byte_enable.size() == 0;
                                     gp.m_byte_enable_length == 0; } );
         finish_item(gp);
      end
      `uvm_info("DATA SEQUENCE", $psprintf("Registradores escritos com
          sucesso") ,UVM_HIGH)
   endtask
endclass

//////////////////////////////////////////////////////////////////////////////
class master_sequence extends uvm_sequence #(uvm_tlm_generic_payload);
   `uvm_object_utils(master_sequence)

   my_sequencer  sequencer_dec;
   my_sequencer  sequencer_wb;

   function new(string name = "");
      super.new(name);
   endfunction

   task body;
```

```
      opcode_sequence op_seq;
      data_sequence  data_seq;

      op_seq     = opcode_sequence::type_id::create("op_seq");
      data_seq   = data_sequence::type_id::create("data_seq");

      forever
      begin
         data_seq.start( sequencer_wb, this );
         assert( op_seq.randomize() );
         op_seq.start( sequencer_dec, this );
      end
   endtask
endclass

//////////////////////////////////////////////////////////////////////////////
class my_driver extends uvm_driver #(uvm_tlm_generic_payload);
   `uvm_component_utils(my_driver)

   uvm_tlm_b_initiator_socket #() isocket;
   uvm_analysis_port #(uvm_tlm_generic_payload) aport;

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      isocket = new("isocket", this);
      aport = new("aport", this);
   endfunction

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
   endfunction

   task run_phase (uvm_phase phase);
      uvm_tlm_generic_payload gp;
      uvm_tlm_time delay = new("del",1e-9);

      `uvm_info("DRIVER", $psprintf("Starting...") , UVM_LOW)
      forever begin
         seq_item_port.get_next_item(gp);
         delay.set_abstime(0,1e-9);
         isocket.b_transport(gp, delay);
         aport.write(gp);
```

```
        if(gp.is_response_error())
           uvm_report_error("DRIVER", $sformatf("ERROR -> Response status:
              %s\n\n", gp.get_response_string));
        else
           `uvm_info("DRIVER", $psprintf("Enviado: %s", gp.convert2string())
              ,UVM_HIGH)
        seq_item_port.item_done();
      end
  endtask
endclass

//////////////////////////////////////////////////////////////////////////////

typedef uvm_sequencer #(uvm_tlm_generic_payload) my_sequencer;

//////////////////////////////////////////////////////////////////////////////
class my_agent extends uvm_agent;

   `uvm_component_utils(my_agent)

   uvm_tlm_b_passthrough_initiator_socket #() agent_pt_socket;
   uvm_analysis_port #(uvm_tlm_generic_payload) aport;

   my_sequencer  my_sequencer_h;
   my_driver     my_driver_h;

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      agent_pt_socket = new("agent_pt_socket", this);
      aport = new("aport", this);
   endfunction

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      my_sequencer_h  = my_sequencer::type_id::create("my_sequencer_h", this);
      my_driver_h     = my_driver::type_id::create("my_driver_h", this);
   endfunction: build_phase

   function void connect_phase(uvm_phase phase);
      my_driver_h.seq_item_port.connect( my_sequencer_h.seq_item_export );
      my_driver_h.isocket.connect( agent_pt_socket );
      my_driver_h.aport.connect( aport );
```

```
   endfunction
endclass

//#############################################################################
class my_monitor extends uvm_component;
   `uvm_component_utils(my_monitor)

   uvm_tlm_b_target_socket #(my_monitor) tsocket;
   uvm_analysis_port #(uvm_tlm_generic_payload) aport;

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      tsocket = new("tsocket", this);
      aport = new("aport", this);
   endfunction

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
   endfunction: build_phase

   task run_phase (uvm_phase phase);
      `uvm_info("MONITOR", $psprintf("Starting...") , UVM_LOW)
   endtask

  virtual task b_transport (uvm_tlm_generic_payload t, uvm_tlm_time delay);
      t.set_response_status(UVM_TLM_OK_RESPONSE);
      `uvm_info("MONITOR", $psprintf("Recebido: %s", t.convert2string())
          ,UVM_HIGH)
      aport.write(t);
  endtask
endclass

//#############################################################################
class my_subscriber extends uvm_subscriber #(uvm_tlm_generic_payload);
   `uvm_component_utils(my_subscriber)

   bit   coverage_achieved = 0;
   int count = 0;
   bit [31:0]  opcode;
   bit [5:0]     op;
   bit [4:0]     rs;
   bit [4:0]     rt;
```

```
covergroup cover_group;
      coverpoint op{
            bins o[] = {[0:63]};
            option.at_least = 1000; }
      coverpoint rs{
            bins s[] = {[0:31]};
            option.at_least = 10000; }
      coverpoint rt{
            bins t[] = {[0:31]};
            option.at_least = 10000; }
endgroup

function new(string name, uvm_component parent=null);
   super.new(name,parent);
   cover_group = new;
endfunction

function void build_phase(uvm_phase phase);
   super.build_phase(phase);
endfunction

task run_phase (uvm_phase phase);
   `uvm_info("SUBSCRIBER", $psprintf("Starting...") , UVM_LOW)
endtask

function void write(uvm_tlm_generic_payload t);
      opcode[7:0]   = t.m_data[0];
      opcode[15:8]  = t.m_data[1];
      opcode[23:16] = t.m_data[2];
      opcode[31:24] = t.m_data[3];

      op = opcode[31:26];
      rs = opcode[25:21];
      rt = opcode[20:16];

      cover_group.sample();
      count++;
      `uvm_info("SUBSCRIBER", $psprintf("Coverage: %0d%%", $get_coverage())
          ,UVM_LOW)
      if($get_coverage() > 99)
            coverage_achieved = 1;
```

```
   endfunction

   function void report_phase(uvm_phase phase);
      uvm_report_info("SUBSCRIBER", $sformatf("Number of Transactions
          collected: %0d", count));
   endfunction
endclass

//#############################################################################
class my_scoreboard extends uvm_scoreboard;
   `uvm_component_utils(my_scoreboard)

   uvm_analysis_export  #(uvm_tlm_generic_payload) xport_in;
   uvm_analysis_export  #(uvm_tlm_generic_payload) xport_ex;
   uvm_analysis_export  #(uvm_tlm_generic_payload) xport_cp0;
   uvm_analysis_imp   #(uvm_tlm_generic_payload, my_scoreboard) imp_data;

   uvm_tlm_analysis_fifo #(uvm_tlm_generic_payload) fifo_in;
   uvm_tlm_analysis_fifo #(uvm_tlm_generic_payload) fifo_ex;
   uvm_tlm_analysis_fifo #(uvm_tlm_generic_payload) fifo_cp0;

   int m_matches = 0;
   int m_mismatches = 0;
   bit erro;
   int i, j;
   byte r[127:0];
   bit [31:0] opcode;

   struct{
      bit [5:0]     op;
      bit [4:0]     rs;
      bit [4:0]     rt;
      bit [4:0]     rd;
      bit [4:0]     re;
      bit [5:0]     func;
      bit [31:0] data_rs;
      bit [31:0] data_rt;
      bit [31:0] opcode;
   }dec_info;

   struct{
```

```
    bit [31:0]  opcode;
    bit [31:0]  data;
    bit [5:0]    op;
    bit [4:0]    rs;
    bit [4:0]    rt;
    bit [4:0]    rd;
    bit [5:0]    func;
}cp0_info;

function new(string name, uvm_component parent=null);
    super.new(name,parent);
endfunction

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    fifo_in    = new("fifo_in", this);
    fifo_ex    = new("fifo_ex", this);
    fifo_cp0 = new("fifo_cp0", this);
    xport_in = new("xport_in", this);
    xport_ex = new("xport_ex", this);
    xport_cp0  = new("xport_cp0", this);
    imp_data = new("imp_data", this);
endfunction

function void connect_phase(uvm_phase phase);
    xport_in.connect( fifo_in.analysis_export );
    xport_ex.connect( fifo_ex.analysis_export );
    xport_cp0.connect( fifo_cp0.analysis_export );
endfunction

task run_phase(uvm_phase phase);
    uvm_tlm_generic_payload gp_in, gp_ex, gp_cp0;
    string s;

    `uvm_info("SCOREBOARD", $psprintf("Starting...") , UVM_LOW)
    forever
    begin
        fifo_in.get(gp_in);
        fifo_ex.get(gp_ex);
        fifo_cp0.get(gp_cp0);
        erro = 0;
```

```
if(gp_in.m_length != 4)
begin
      $sformat(s, "OPCODE Length does not match");
      uvm_report_error("FIFO Error", s);
      erro = 1;
end
if(gp_ex.m_length != 48)
begin
      $sformat(s, "DEC_INFO Length does not match");
      uvm_report_error("FIFO Error", s);
      erro = 1;
end
if(gp_cp0.m_length != 32)
begin
      $sformat(s, "COP0_INFO Length does not match");
      uvm_report_error("FIFO Error", s);
      erro = 1;
end

opcode[7:0]   = gp_in.m_data[0];
opcode[15:8]  = gp_in.m_data[1];
opcode[23:16] = gp_in.m_data[2];
opcode[31:24] = gp_in.m_data[3];

dec_info.op = gp_ex.m_data[0*4][5:0];
dec_info.rs = gp_ex.m_data[1*4][4:0];
dec_info.rt = gp_ex.m_data[2*4][4:0];
dec_info.rd = gp_ex.m_data[3*4][4:0];
dec_info.re = gp_ex.m_data[4*4][4:0];
dec_info.func  = gp_ex.m_data[5*4][5:0];
dec_info.data_rs[7:0]  = gp_ex.m_data[9*4];
dec_info.data_rs[15:8] = gp_ex.m_data[9*4+1];
dec_info.data_rs[23:16] = gp_ex.m_data[9*4+2];
dec_info.data_rs[31:24] = gp_ex.m_data[9*4+3];
dec_info.data_rt[7:0]  = gp_ex.m_data[10*4];
dec_info.data_rt[15:8] = gp_ex.m_data[10*4+1];
dec_info.data_rt[23:16] = gp_ex.m_data[10*4+2];
dec_info.data_rt[31:24] = gp_ex.m_data[10*4+3];
dec_info.opcode[7:0] = gp_ex.m_data[11*4];
dec_info.opcode[15:8] = gp_ex.m_data[11*4+1];
dec_info.opcode[23:16] = gp_ex.m_data[11*4+2];
dec_info.opcode[31:24] = gp_ex.m_data[11*4+3];
```

```
cp0_info.opcode[7:0] = gp_cp0.m_data[0];
cp0_info.opcode[15:8] = gp_cp0.m_data[1];
cp0_info.opcode[23:16] = gp_cp0.m_data[2];
cp0_info.opcode[31:24] = gp_cp0.m_data[3];
cp0_info.data[7:0]  = gp_cp0.m_data[1*4];
cp0_info.data[15:8] = gp_cp0.m_data[1*4+1];
cp0_info.data[23:16] = gp_cp0.m_data[1*4+2];
cp0_info.data[31:24] = gp_cp0.m_data[1*4+3];
cp0_info.op     = gp_cp0.m_data[2*4][5:0];
cp0_info.rs     = gp_cp0.m_data[3*4][4:0];
cp0_info.rt     = gp_cp0.m_data[4*4][4:0];
cp0_info.rd     = gp_cp0.m_data[5*4][4:0];
cp0_info.func   = gp_cp0.m_data[6*4][5:0];

if(dec_info.op != opcode[31:26])
begin
     $sformat(s, "OP does not match");
     uvm_report_error("Comparator Decode Mismatch", s);
     erro = 1;
end
if(dec_info.rs != opcode[25:21])
begin
     $sformat(s, "RS does not match");
     uvm_report_error("Comparator Decode Mismatch", s);
     erro = 1;
end
if(dec_info.rt != opcode[20:16])
begin
     $sformat(s, "RT does not match");
     uvm_report_error("Comparator Decode Mismatch", s);
     erro = 1;
end
if(dec_info.rd != opcode[15:11])
begin
     $sformat(s, "RD does not match");
     uvm_report_error("Comparator Decode Mismatch", s);
     erro = 1;
end
if(dec_info.re != opcode[10:6])
begin
     $sformat(s, "RE does not match");
```

```
      uvm_report_error("Comparator Decode Mismatch", s);
      erro = 1;
end
if(dec_info.func != opcode[5:0])
begin
      $sformat(s, "FUNC does not match");
      uvm_report_error("Comparator Decode Mismatch", s);
      erro = 1;
end
if( (dec_info.data_rs[7:0]  != r[dec_info.rs*4]   ||
      dec_info.data_rs[15:8]  != r[dec_info.rs*4+1] ||
      dec_info.data_rs[23:16] != r[dec_info.rs*4+2] ||
      dec_info.data_rs[31:24] != r[dec_info.rs*4+3]) &&
      (dec_info.rs == 0 && dec_info.data_rs != 0 ) )
begin
      $sformat(s, "DATA_RS does not match");
      uvm_report_error("Comparator Decode Mismatch", s);
      erro = 1;
end
if( (dec_info.data_rt[7:0]  != r[dec_info.rt*4]   ||
      dec_info.data_rt[15:8]  != r[dec_info.rt*4+1] ||
      dec_info.data_rt[23:16] != r[dec_info.rt*4+2] ||
      dec_info.data_rt[31:24] != r[dec_info.rt*4+3]) &&
      (dec_info.rt == 0 && dec_info.data_rt != 0 ) )
begin
      $sformat(s, "DATA_RT does not match");
      uvm_report_error("Comparator Decode Mismatch", s);
      erro = 1;
end
if(dec_info.opcode != opcode)
begin
      $sformat(s, "OPCODE does not match");
      uvm_report_error("Comparator Decode Mismatch", s);
      erro = 1;
end


if(cp0_info.opcode != opcode)
begin
      $sformat(s, "OPCODE does not match");
      uvm_report_error("Comparator COP0 Mismatch", s);
      erro = 1;
```

```
end
if( (cp0_info.data[7:0] != r[cp0_info.rt*4]  ||
     cp0_info.data[15:8] != r[cp0_info.rt*4+1] ||
     cp0_info.data[23:16] != r[cp0_info.rt*4+2] ||
     cp0_info.data[31:24] != r[cp0_info.rt*4+3]) &&
     (cp0_info.rt == 0 && cp0_info.data != 0 ) )
begin
     $sformat(s, "DATA does not match");
     uvm_report_error("Comparator COP0 Mismatch: ", s);
     erro = 1;
end
if(cp0_info.op != opcode[31:26])
begin
     $sformat(s, "OP does not match");
     uvm_report_error("Comparator COP0 Mismatch: ", s);
     erro = 1;
end
if(cp0_info.rs != opcode[25:21])
begin
     $sformat(s, "RS does not match");
     uvm_report_error("Comparator COP0 Mismatch: ", s);
     erro = 1;
end
if(cp0_info.rt != opcode[20:16])
begin
     $sformat(s, "RT does not match");
     uvm_report_error("Comparator COP0 Mismatch: ", s);
     erro = 1;
end
if(cp0_info.rd != opcode[15:11])
begin
     $sformat(s, "RD does not match");
     uvm_report_error("Comparator COP0 Mismatch: ", s);
     erro = 1;
end
if(cp0_info.func != opcode[5:0])
begin
     $sformat(s, "FUNC does not match");
     uvm_report_error("Comparator COP0 Mismatch: ", s);
     erro = 1;
end
```

```
        if(erro == 1)
           m_mismatches++;
        else
           m_matches++;
     end
  endtask

  function void report_phase(uvm_phase phase);
     uvm_report_info("SCOREBOARD", $sformatf("Matches: %0d", m_matches));
     uvm_report_info("SCOREBOARD", $sformatf("Mismatches: %0d",
         m_mismatches));
  endfunction

  function void write(uvm_tlm_generic_payload t);
  int i=0;
     for(int j=0; j<4; j++)
     begin
        r[i]= t.m_data[j];
        i++;
     end
  endfunction
endclass

//#############################################################################
class my_env extends uvm_env;
  `uvm_component_utils(my_env)

  uvm_tlm_b_passthrough_initiator_socket #() env_dec_pt_socket;
  uvm_tlm_b_passthrough_initiator_socket #() env_wb_pt_socket;
  uvm_tlm_b_passthrough_target_socket #() env_ex_pt_socket;
  uvm_tlm_b_passthrough_target_socket #() env_cp0_pt_socket;

  my_agent      agent_wb;
  my_agent      agent_dec;
  my_monitor    monitor_ex;
  my_monitor    monitor_cp0;
  my_subscriber subscriber_h;
  my_scoreboard scoreboard_h;

  function new(string name, uvm_component parent=null);
     super.new(name,parent);
     env_wb_pt_socket = new("env_wb_pt_socket", this);
```

```
      env_dec_pt_socket = new("env_dec_pt_socket", this);
      env_ex_pt_socket = new("env_ex_pt_socket", this);
      env_cp0_pt_socket = new("env_cp0_pt_socket", this);
   endfunction

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      agent_dec = my_agent::type_id::create("agent_dec", this);
      agent_wb   = my_agent::type_id::create("agent_wb", this);
      monitor_ex  = my_monitor::type_id::create("monitor_ex", this);
      monitor_cp0 = my_monitor::type_id::create("monitor_cp0", this);
      subscriber_h = my_subscriber::type_id::create("subscriber_h", this);
      scoreboard_h = my_scoreboard::type_id::create("scoreboard_h", this);
   endfunction

   function void connect_phase(uvm_phase phase);
      agent_dec.agent_pt_socket.connect ( env_dec_pt_socket );
      agent_wb.agent_pt_socket.connect ( env_wb_pt_socket );
      env_ex_pt_socket.connect ( monitor_ex.tsocket );
      env_cp0_pt_socket.connect ( monitor_cp0.tsocket );
      agent_dec.aport.connect( subscriber_h.analysis_export );
      agent_wb.aport.connect ( scoreboard_h.imp_data );
      agent_dec.aport.connect ( scoreboard_h.xport_in );
      monitor_ex.aport.connect ( scoreboard_h.xport_ex );
      monitor_cp0.aport.connect( scoreboard_h.xport_cp0 );
   endfunction
endclass

//////////////////////////////////////////////////////////////////////////////
class my_test extends uvm_test;
   `uvm_component_utils(my_test)

   uvm_tlm_b_passthrough_initiator_socket #() test_dec_pt_socket;
   uvm_tlm_b_passthrough_initiator_socket #() test_wb_pt_socket;
   uvm_tlm_b_passthrough_target_socket #() test_ex_pt_socket;
   uvm_tlm_b_passthrough_target_socket #() test_cp0_pt_socket;
   my_env          my_env_h;

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      test_dec_pt_socket = new("test_dec_pt_socket", this);
      test_wb_pt_socket = new("test_wb_pt_socket", this);
```

```
      test_ex_pt_socket = new("test_ex_pt_socket", this);
      test_cp0_pt_socket = new("test_cp0_pt_socket", this);
   endfunction

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      my_env_h      = my_env::type_id::create("my_env_h", this);
   endfunction

   function void connect_phase(uvm_phase phase);
      my_env_h.env_dec_pt_socket.connect( test_dec_pt_socket );
      my_env_h.env_wb_pt_socket.connect( test_wb_pt_socket );
      test_ex_pt_socket.connect( my_env_h.env_ex_pt_socket );
      test_cp0_pt_socket.connect( my_env_h.env_cp0_pt_socket );
   endfunction

   task run_phase(uvm_phase phase);
      master_sequence seq;
      seq = master_sequence::type_id::create("seq");

      phase.raise_objection(this);
      `uvm_info("TEST", $psprintf("Starting...") , UVM_LOW)
      seq.sequencer_dec = my_env_h.agent_dec.my_sequencer_h;
      seq.sequencer_wb  = my_env_h.agent_wb.my_sequencer_h;
      assert( seq.randomize() );

      fork
         begin
            @(posedge my_env_h.subscriber_h.coverage_achieved);
         end
         begin
            seq.start(null);
         end
      join_any
      `uvm_info("TEST", $psprintf("Coverage Goals have been achieved!") ,
         UVM_LOW)
      `uvm_info("TEST", $psprintf("Ending test and dropping objection...") ,
         UVM_LOW)
      phase.drop_objection(this);
   endtask
endclass
```

ANEXO E – *Log* de Simulação

```
Loading snapshot worklib.sv_main:sv ..................... Done
SVSEED default: 1
Connecting an SC-side proxy port for 'dec.f_tsocket' with lookup string 'fetch'
     for later connection with SV
Connecting an SC-side proxy port for 'dec.wb_tsocket' with lookup string '
    writeback' for later connection with SV
Connecting an SC-side proxy chan for 'dec.e_isocket' with lookup string '
    execute' for later connection with SV
Connecting an SC-side proxy chan for 'dec.cp0_isocket' with lookup string 'cop0
    ' for later connection with SV
ncsim> source /quartzo/cadence/linux/INCISIV/tools/inca/files/ncsimrc
ncsim> source /quartzo/cadence/linux/INCISIV/tools/uvm/uvm_lib/uvm_sv/files/tcl
    /uvm_sim.tcl
ncsim> run
ncsim: *W,COVUSC: Unsupported SystemC module (worklib.sc_main) specified for
    coverage.
ncsim: *W,COVNIB: Currently, by default, coverage is scored for implicit else
    and case default blocks. In subsequent releases, the default scoring for
    such blocks will be disabled and a coverage configuration file command will
     be provided to revert to the old behavior. This change may prevent
    refinements from 13.1 releases to be applied on coverage databases
    generated using subsequent releases in the default mode..
----------------------------------------------------------------
UVM-1.1d
(C) 2007-2013 Mentor Graphics Corporation
(C) 2007-2013 Cadence Design Systems, Inc.
(C) 2006-2013 Synopsys, Inc.
(C) 2011-2013 Cypress Semiconductor Corp.
----------------------------------------------------------------

  ***********      IMPORTANT RELEASE NOTES      ************

  You are using a version of the UVM library that has been compiled
  with `UVM_NO_DEPRECATED undefined.
  See http://www.eda.org/svdb/view.php?id=3313 for more details.

  You are using a version of the UVM library that has been compiled
  with `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR undefined.
```

```
  See http://www.eda.org/svdb/view.php?id=3770 for more details.

----------------------------------------------------------------
UVMC-2.2
(C) 2009-2012 Mentor Graphics Corporation
----------------------------------------------------------------
Registering SV-side 'teste.test_dec_pt_socket' and lookup string 'fetch' for
    later connection with SC
Registering SV-side 'teste.test_wb_pt_socket' and lookup string 'writeback' for
     later connection with SC
Registering SV-side 'teste.test_ex_pt_socket' and lookup string 'execute' for
    later connection with SC
Registering SV-side 'teste.test_cp0_pt_socket' and lookup string 'cop0' for
    later connection with SC
UVM_INFO @ 0: reporter [RNTST] Running test ...
Connected SC-side 'dec.cp0_isocket' to SV-side 'teste.test_cp0_pt_socket'
Connected SC-side 'dec.e_isocket' to SV-side 'teste.test_ex_pt_socket'
Connected SC-side 'dec.f_tsocket' to SV-side 'teste.test_dec_pt_socket'
Connected SC-side 'dec.wb_tsocket' to SV-side 'teste.test_wb_pt_socket'
UVM_INFO ./common/testbench.sv(693) @ 0: teste [TEST] Starting...
UVM_INFO ./common/testbench.sv(202) @ 0: teste.my_env_h.monitor_ex [MONITOR]
    Starting...
UVM_INFO ./common/testbench.sv(202) @ 0: teste.my_env_h.monitor_cp0 [MONITOR]
    Starting...
UVM_INFO ./common/testbench.sv(120) @ 0: teste.my_env_h.agent_wb.my_driver_h [
    DRIVER] Starting...
UVM_INFO ./common/testbench.sv(120) @ 0: teste.my_env_h.agent_dec.my_driver_h [
    DRIVER] Starting...
UVM_INFO ./common/testbench.sv(702) @ 0: teste [TEST] Data Sequence finished
    his job
UVM_INFO ./common/testbench.sv(703) @ 0: teste [TEST] All register were
    succesfull writen
UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 0%
(...)
UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 44%
(...)
UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 60%
(...)
```

```
UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 90%
(...)
UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 99%
UVM_INFO ./common/testbench.sv(268) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 100%
UVM_INFO ./common/testbench.sv(716) @ 0: teste [TEST] Coverage Goals have been
    achieved!
UVM_INFO ./common/testbench.sv(717) @ 0: teste [TEST] Ending test and drop
    objection...
UVM_INFO /home/alunos/junior_tcc2/uvm-1.1d/src/base/uvm_objection.svh(1268) @
    0: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract'
    phase
UVM_INFO ./common/testbench.sv(571) @ 0: teste.my_env_h.scoreboard_h [
    SCOREBOARD] Matches:  328238
UVM_INFO ./common/testbench.sv(572) @ 0: teste.my_env_h.scoreboard_h [
    SCOREBOARD] Mismatches: 0
UVM_INFO ./common/testbench.sv(281) @ 0: teste.my_env_h.subscriber_h [COVERAGE
    COLLECTOR] Number of Transactions: 328238

--- UVM Report catcher Summary ---


Number of demoted UVM_FATAL reports : 0
Number of demoted UVM_ERROR reports : 0
Number of demoted UVM_WARNING reports: 0
Number of caught UVM_FATAL reports :  0
Number of caught UVM_ERROR reports :  0
Number of caught UVM_WARNING reports : 0

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :328252
UVM_WARNING :   0
UVM_ERROR :   0
UVM_FATAL :   0
** Report counts by id
[COVERAGE COLLECTOR]   1
[DRIVER]    2
[MONITOR]    2
```

```
[RNTST]    1
[SCOREBOARD]    2
[SUBSCRIBER] 328238
[TEST]    5
[TEST_DONE]    1
Simulation complete via $finish(1) at time 0 FS + 984888
/home/alunos/junior_tcc2/uvm-1.1d/src/base/uvm_root.svh:430 $finish;
ncsim> exit
ncsim: *W,CGPIZE: Instance coverage for covergroup instance "cover_group" will
    not be dumped to database as per_instance option value is set to 0:./common
    /testbench.sv, 228.

coverage setup:
  workdir :  ./cov_work
  dutinst :  sc_main(sc_main)
  dutinst :  sv_main(sv_main)
  scope   :  scope
  testname : test

coverage files:
  model(design data) : ./cov_work/scope/icc_542f9204_41b44a0c.ucm
  data               :  ./cov_work/scope/test/icc_542f9204_41b44a0c.ucd
```

ANEXO F – *Log* de Simulação com Erros Embutidos

```
Loading snapshot worklib.sv_main:sv ..................... Done
SVSEED default: 1
Connecting an SC-side proxy port for 'dec.f_tsocket' with lookup string 'fetch'
     for later connection with SV
Connecting an SC-side proxy port for 'dec.wb_tsocket' with lookup string '
    writeback' for later connection with SV
Connecting an SC-side proxy chan for 'dec.e_isocket' with lookup string '
    execute' for later connection with SV
Connecting an SC-side proxy chan for 'dec.cp0_isocket' with lookup string 'cop0
    ' for later connection with SV
ncsim> source /quartzo/cadence/linux/INCISIV/tools/inca/files/ncsimrc
ncsim> source /quartzo/cadence/linux/INCISIV/tools/uvm/uvm_lib/uvm_sv/files/tcl
    /uvm_sim.tcl
ncsim> run
ncsim: *W,COVUSC: Unsupported SystemC module (worklib.sc_main) specified for
    coverage.
ncsim: *W,COVNIB: Currently, by default, coverage is scored for implicit else
    and case default blocks. In subsequent releases, the default scoring for
    such blocks will be disabled and a coverage configuration file command will
     be provided to revert to the old behavior. This change may prevent
    refinements from 13.1 releases to be applied on coverage databases
    generated using subsequent releases in the default mode..
----------------------------------------------------------------
UVM-1.1d
(C) 2007-2013 Mentor Graphics Corporation
(C) 2007-2013 Cadence Design Systems, Inc.
(C) 2006-2013 Synopsys, Inc.
(C) 2011-2013 Cypress Semiconductor Corp.
----------------------------------------------------------------

  ***********      IMPORTANT RELEASE NOTES      ************

  You are using a version of the UVM library that has been compiled
  with `UVM_NO_DEPRECATED undefined.
  See http://www.eda.org/svdb/view.php?id=3313 for more details.
```

```
  You are using a version of the UVM library that has been compiled
  with 'UVM_OBJECT_MUST_HAVE_CONSTRUCTOR undefined.
  See http://www.eda.org/svdb/view.php?id=3770 for more details.

----------------------------------------------------------------
UVMC-2.2
(C) 2009-2012 Mentor Graphics Corporation
----------------------------------------------------------------
Registering SV-side 'teste.test_dec_pt_socket' and lookup string 'fetch' for
    later connection with SC
Registering SV-side 'teste.test_wb_pt_socket' and lookup string 'writeback' for
     later connection with SC
Registering SV-side 'teste.test_ex_pt_socket' and lookup string 'execute' for
    later connection with SC
Registering SV-side 'teste.test_cp0_pt_socket' and lookup string 'cop0' for
    later connection with SC
UVM_INFO @ 0: reporter [RNTST] Running test ...
Connected SC-side 'dec.cp0_isocket' to SV-side 'teste.test_cp0_pt_socket'
Connected SC-side 'dec.e_isocket' to SV-side 'teste.test_ex_pt_socket'
Connected SC-side 'dec.f_tsocket' to SV-side 'teste.test_dec_pt_socket'
Connected SC-side 'dec.wb_tsocket' to SV-side 'teste.test_wb_pt_socket'
UVM_INFO ./common/testbench.sv(724) @ 0: teste [TEST] Starting...
UVM_INFO ./common/testbench.sv(243) @ 0: teste.my_env_h.monitor_ex [MONITOR]
    Starting...
UVM_INFO ./common/testbench.sv(243) @ 0: teste.my_env_h.monitor_cp0 [MONITOR]
    Starting...
UVM_INFO ./common/testbench.sv(159) @ 0: teste.my_env_h.agent_wb.my_driver_h [
    DRIVER] Starting...
UVM_INFO ./common/testbench.sv(159) @ 0: teste.my_env_h.agent_dec.my_driver_h [
    DRIVER] Starting...
UVM_INFO ./common/testbench.sv(312) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 0%
(...)
UVM_INFO ./common/testbench.sv(312) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 99%
UVM_ERROR @ 0: teste.my_env_h.scoreboard_h [Comparator Decode Mismatch] RD does
     not match
UVM_ERROR @ 0: teste.my_env_h.scoreboard_h [Comparator COP0 Mismatch: ] RD does
     not match
UVM_INFO ./common/testbench.sv(312) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 99%
```

```
UVM_INFO ./common/testbench.sv(312) @ 0: teste.my_env_h.subscriber_h [
    SUBSCRIBER] Coverage: 100%
UVM_INFO ./common/testbench.sv(741) @ 0: teste [TEST] Coverage Goals have been
    achieved!
UVM_INFO ./common/testbench.sv(742) @ 0: teste [TEST] Ending test and drop
    objection...
UVM_INFO /home/alunos/junior_tcc2/uvm-1.1d/src/base/uvm_objection.svh(1268) @
    0: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract'
    phase
UVM_INFO @ 0: teste.my_env_h.scoreboard_h [SCOREBOARD] Matches: 240
UVM_INFO @ 0: teste.my_env_h.scoreboard_h [SCOREBOARD] Mismatches: 7546
UVM_INFO @ 0: teste.my_env_h.subscriber_h [SUBSCRIBER] Number of Transactions
    collected: 7786

--- UVM Report catcher Summary ---


Number of demoted UVM_FATAL reports : 0
Number of demoted UVM_ERROR reports : 0
Number of demoted UVM_WARNING reports: 0
Number of caught UVM_FATAL reports :  0
Number of caught UVM_ERROR reports :  0
Number of caught UVM_WARNING reports : 0

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO : 7801
UVM_WARNING :  0
UVM_ERROR :15092
UVM_FATAL :  0
** Report counts by id
[Comparator COP0 Mismatch: ] 7546
[Comparator Decode Mismatch] 7546
[DRIVER]    4
[MONITOR]    2
[RNTST]    1
[SCOREBOARD]    2
[SUBSCRIBER] 7788
[TEST]    3
[TEST_DONE]    1
Simulation complete via $finish(1) at time 0 FS + 27170
```

```
/home/alunos/junior_tcc2/uvm-1.1d/src/base/uvm_root.svh:430 $finish;
ncsim> exit
ncsim: *W,CGPIZE: Instance coverage for covergroup instance "cover_group" will
    not be dumped to database as per_instance option value is set to 0:./common
    /testbench.sv, 269.

coverage setup:
  workdir :  ./cov_work
  dutinst :  sc_main(sc_main)
  dutinst :  sv_main(sv_main)
  scope   :  scope
  testname : test

coverage files:
  model(design data) : ./cov_work/scope/icc_542f9204_45f5e0df.ucm (reused)
  data               :  ./cov_work/scope/test/icc_542f9204_45f5e0df.ucd
```